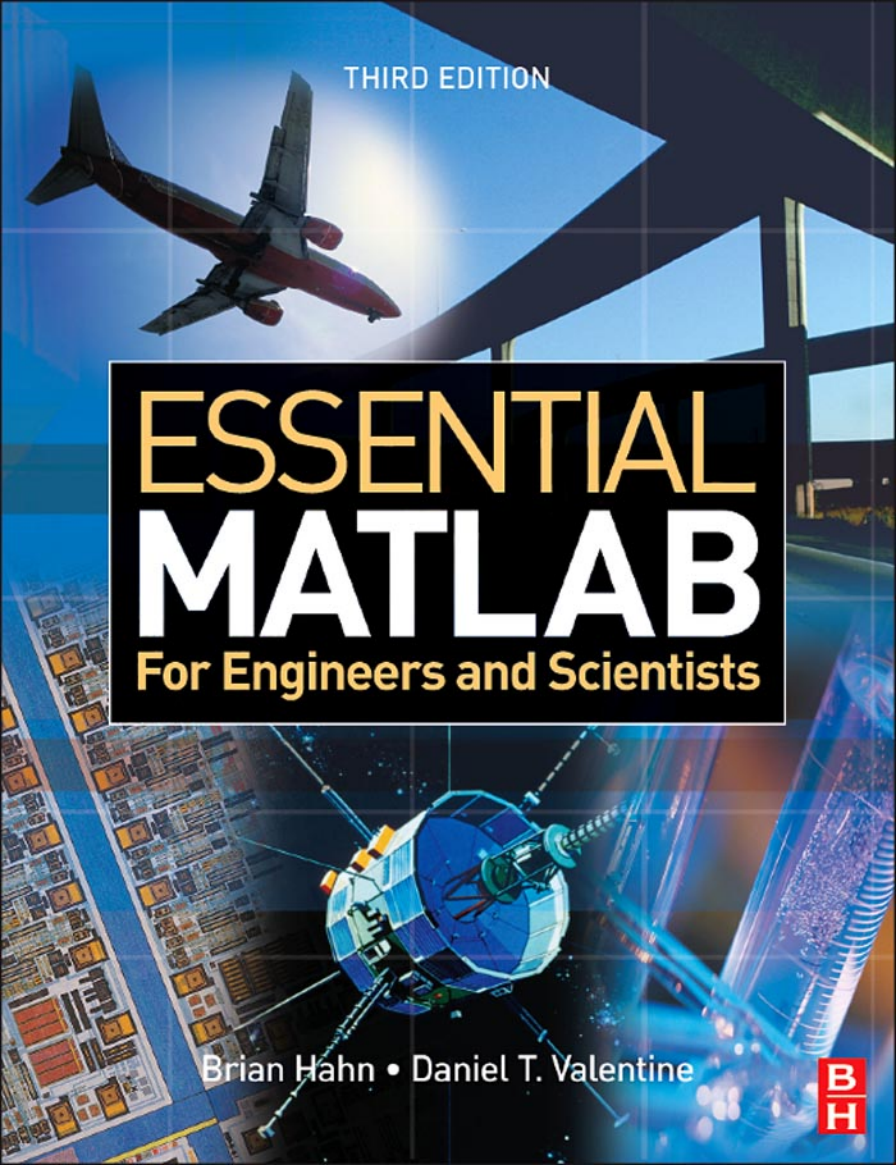
VISIT…


THIRD EDITION
ESSENTIAL
MATLAB
For Engineers and Scientists
Brian Hahn • Daniel T. Valentine
B
H

Essential MATLAB® for Engineers and Scientists

Reviewers' Quotes

This book provides an excellent initiation into programming in MATLAB while serving as a teaser for more advanced topics. It provides a structured entry into MATLAB programming through well designed exercises.

Carl H. Sondergeld
Professor and Curtis Mewbourne Chair
Mewbourne School of Petroleum and Geological Engineering
University of Oklahoma

This updated version continues to provide beginners with the essentials of Matlab, with many examples from science and engineering, written in an informal and accessible style. The new chapter on algorithm development and program design provides an excellent introduction to a structured approach to problem solving and the use of MATLAB as a programming language.

Professor Gary Ford
Department of Electrical and Computer Engineering
University of California, Davis

For a while I have been searching for a good MATLAB text for a graduate course on methods in environmental sciences. I finally settled on Hahn and Valentine because it provides the balance I need regarding ease of use and relevance of material and examples.

Professor Wayne M. Getz
Department Environmental Science Policy & Management
University of California at Berkeley

This book is an outstanding introductory text for teaching mathematics, engineering, and science students how MATLAB can be used to solve mathematical problems. Its intuitive and well-chosen examples nicely bridge the gap between prototypical mathematical models and how MATLAB can be used to evaluate these models. The author does a superior job of examining and explaining the MATLAB code used to solve the problems presented.

Professor Mark E. Cawood
Department of Mathematical Sciences
Clemson University

Essential MATLAB® for Engineers and Scientists

Third edition

Brian D. Hahn
and
Daniel T. Valentine

AMSTERDAM • BOSTON • HEIDELBERG • LONDON • NEW YORK • OXFORD
PARIS • SAN DIEGO • SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Butterworth-Heinemann is an imprint of Elsevier

Butterworth-Heinemann is an imprint of Elsevier
Linacre House, Jordan Hill, Oxford, OX2 8DP
30 Corporate Drive, Burlington, MA 01803

First published 2002
Reprinted 2002, 2003, 2004, 2005, 2006
Second edition 2006
Third edition 2007

Copyright © 2002, 2006, 2007 Brian D. Hahn and Daniel T. Valentine. Published by Elsevier Ltd. All rights reserved

The right of Brian D. Hahn and Daniel T. Valentine to be identified as the authors of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means electronic, mechanical, photocopying, recording or otherwise without the prior written permission of the publisher

Permission may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone (+44) (0) 1865 843830; fax (+44) (0) 1865 853333; email: permissions@elsevier.com. Alternatively you can submit your request online by visiting the Elsevier web site at http://elsevier.com/locate/permissions, and selecting *Obtaining permission to use Elsevier material*

Notice
No responsibility is assumed by the publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein. Because of rapid advances in the medical sciences, in particular, independent verification of diagnoses and drug dosages should be made

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

ISBN 13: 9-78-0-75-068417-0
ISBN 10: 0-75-068417-8

For information on all Butterworth-Heinemann publications visit our web site at books.elsevier.com

Typeset by Charon Tec Ltd (A Macmillan Company), Chennai, India,
www.charontec.com
Printed and bound in Italy

07 08 09 10 11 10 9 8 7 6 5 4 3 2 1

Working together to grow libraries in developing countries
www.elsevier.com | www.bookaid.org | www.sabre.org
ELSEVIER BOOK AID International Sabre Foundation

Contents

Preface to the third edition xvii

PART I ESSENTIALS 1

1 Introduction 3
1.1 Using MATLAB 4
1.2 The MATLAB desktop 15
1.3 Sample program 16
1.3.1 Cut and paste 16
1.3.2 Saving a program: script files 19
1.3.3 How a program works 21

2 MATLAB fundamentals 24
2.1 Variables and the workspace 24
2.1.1 Variables 24
2.1.2 Case sensitivity 25
2.1.3 The workspace 25
2.1.4 Adding commonly used constants to the workspace 27
2.2 Arrays: vectors and matrices 27
2.2.1 Initializing vectors: explicit lists 28
2.2.2 Initializing vectors: the colon operator 29
2.2.3 `linspace` 30
2.2.4 Transposing vectors 30
2.2.5 Subscripts 31
2.2.6 Matrices 31
2.2.7 Capturing output 32
2.3 Vertical motion under gravity 33
2.4 Operators, expressions and statements 35
2.4.1 Numbers 35
2.4.2 Data types 36
2.4.3 Arithmetic operators 37
2.4.4 Precedence of operators 37
2.4.5 The colon operator 38
2.4.6 The transpose operator 39

2.4.7 Arithmetic operations on arrays 39
2.4.8 Expressions 41
2.4.9 Statements 41
2.4.10 Statements, commands and functions 43
2.4.11 Vectorization of formulae 43
2.5 Output 47
2.5.1 `disp` 47
2.5.2 `format` 49
2.5.3 Scale factors 50
2.6 Repeating with `for` 51
2.6.1 Square roots with Newton's method 51
2.6.2 Factorials! 53
2.6.3 Limit of a sequence 53
2.6.4 The basic `for` construct 54
2.6.5 `for` in a single line 56
2.6.6 More general `for` 56
2.6.7 Avoid `for` loops by vectorizing! 56
2.6.8 A common mistake: `for` less loops! 59
2.7 Decisions 60
2.7.1 The one-line `if` statement 60
2.7.2 The `if-else` construct 62
2.7.3 The one-line `if-else` statement 63
2.7.4 `elseif` 64
2.7.5 Logical operators 65
2.7.6 Multiple `if`s versus `elseif` 65
2.7.7 Nested `if`s 67
2.7.8 Vectorizing `if`s? 68
2.7.9 `switch` 68
2.8 Complex numbers 69
2.9 More on input and output 71
2.9.1 `fprintf` 71
2.9.2 Output to a disk file with `fprintf` 73
2.9.3 General file I/O 73
2.9.4 Saving and loading data 73
2.10 Odds 'n ends 73
2.10.1 Variables, functions and scripts with the same name 73
2.10.2 The `input` statement 74
2.10.3 Shelling out to the operating system 75
2.10.4 More Help functions 76
2.11 Programming style 76

3 Program design and algorithm development 86
3.1 Computer program design process 87
3.1.1 Projectile problem example 89

3.2 Other examples of structure plans 98
3.2.1 Quadratic equation 99
3.3 Structured programming with functions 100

4 MATLAB functions & *data import-export utilities 104
4.1 Some common functions 105
4.2 *Importing and exporting data 110
4.2.1 The `load` and `save` commands 110
4.2.2 Exporting text (ASCII) data 110
4.2.3 Importing text (ASCII) data 111
4.2.4 Exporting binary data 111
4.2.5 The Import Wizard 112
4.2.6 Low-level file I/O functions 113
4.2.7 Other import/export functions 118

5 Logical vectors 121
5.1 Examples 122
5.1.1 Discontinuous graphs 122
5.1.2 Avoiding division by zero 123
5.1.3 Avoiding infinity 125
5.1.4 Counting random numbers 126
5.1.5 Rolling dice 127
5.2 Logical operators 127
5.2.1 Operator precedence 129
5.2.2 Danger 130
5.2.3 Logical operators and vectors 130
5.3 Subscripting with logical vectors 131
5.4 Logical functions 133
5.4.1 Using `any` and `all` 134
5.5 Logical vectors instead of `elseif` ladders 135

6 Matrices of numbers & arrays of strings 141
6.1 Matrices 142
6.1.1 A concrete example 142
6.1.2 Creating matrices 143
6.1.3 Subscripts 144
6.1.4 Transpose 144
6.1.5 The colon operator 144
6.1.6 Duplicating rows and columns: tiling 148
6.1.7 Deleting rows and columns 148
6.1.8 Elementary matrices 149
6.1.9 *Specialized matrices 150
6.1.10 Using MATLAB functions with matrices 151
6.1.11 Manipulating matrices 152

6.1.12 Array (element-by-element) operations on matrices 153
6.1.13 Matrices and `for` 153
6.1.14 Visualization of matrices 154
6.1.15 Vectorizing nested `for`s: loan repayment tables 154
6.1.16 Multidimensional arrays 156
6.2 Matrix operations 157
6.2.1 Matrix multiplication 157
6.2.2 Matrix exponentiation 159
6.3 Other matrix functions 160
6.4 *Strings 160
6.4.1 Assignment 160
6.4.2 Input 160
6.4.3 Strings are arrays 161
6.4.4 Concatenation of strings 161
6.4.5 ASCII codes, `double` and `char` 162
6.4.6 `fprintf` of strings 163
6.4.7 Comparing strings 163
6.4.8 Other string functions 164
6.5 *Two-dimensional strings 164
6.6 *`eval` and text macros 165
6.6.1 Error trapping with `eval` and `lasterr` 166
6.6.2 `eval` with `try...catch` 167

7 Introduction to graphics 171
7.1 Basic 2-D graphs 171
7.1.1 Labels 173
7.1.2 Multiple plots on the same axes 173
7.1.3 Line styles, markers and color 174
7.1.4 Axis limits 175
7.1.5 Multiple plots in a figure: `subplot` 176
7.1.6 `figure`, `clf` and `cla` 178
7.1.7 Graphical input 178
7.1.8 Logarithmic plots 178
7.1.9 Polar plots 179
7.1.10 Plotting rapidly changing mathematical functions: `fplot` 180
7.1.11 The property editor 181
7.2 3-D plots 181
7.2.1 `plot3` 182
7.2.2 Animated 3-D plots with `comet3` 183
7.2.3 Mesh surfaces 183
7.2.4 Contour plots 186
7.2.5 Cropping a surface with `NaN`s 187
7.2.6 Visualizing vector fields 188
7.2.7 Visualization of matrices 189

7.2.8 Rotation of 3-D graphs 190
7.2.9 Other cool graphics functions 192

8 Loops 205
8.1 Determinate repetition with `for` 205
8.1.1 Binomial coefficient 205
8.1.2 Update processes 206
8.1.3 Nested `for`s 208
8.2 Indeterminate repetition with `while` 208
8.2.1 A guessing game 208
8.2.2 The `while` statement 209
8.2.3 Doubling time of an investment 210
8.2.4 Prime numbers 211
8.2.5 Projectile trajectory 212
8.2.6 `break` and `continue` 215
8.2.7 Menus 215

9 Errors and pitfalls 222
9.1 Syntax errors 222
9.1.1 `lasterr` 225
9.2 Pitfalls and surprises 225
9.2.1 Incompatible vector sizes 225
9.2.2 Name hiding 225
9.2.3 Other pitfalls for the unwary 226
9.3 Errors in logic 226
9.4 Rounding error 226
9.5 Trapping and generating errors 228

10 Function M-files 230
10.1 Some examples 230
10.1.1 Inline objects: harmonic oscillators 230
10.1.2 Function M-files: Newton's method again 232
10.2 Basic rules 233
10.2.1 Subfunctions 239
10.2.2 Private functions 239
10.2.3 P-code files 239
10.2.4 Improving M-file performance with the profiler 240
10.3 Function handles 240
10.4 Command/function duality 242
10.5 Function name resolution 243
10.6 Debugging M-files 243
10.6.1 Debugging a script 244
10.6.2 Debugging a function 246
10.7 Recursion 246

11 Vectors as arrays & *advanced data structures 251
11.1 Update processes 251
11.1.1 Unit time steps 252
11.1.2 Non-unit time steps 255
11.1.3 Using a function 256
11.1.4 Exact solution 258
11.2 Frequencies, bar charts and histograms 259
11.2.1 A random walk 259
11.2.2 Histograms 260
11.3 *Sorting 261
11.3.1 Bubble Sort 261
11.3.2 MATLAB's sort 263
11.4 *Structures 264
11.5 *Cell arrays 267
11.5.1 Assigning data to cell arrays 267
11.5.2 Accessing data in cell arrays 268
11.5.3 Using cell arrays 269
11.5.4 Displaying and visualizing cell arrays 270
11.6 *Classes and objects 270

12 *More graphics 272
12.1 Handle Graphics 272
12.1.1 Getting handles 273
12.1.2 Graphics object properties and how to change them 274
12.1.3 A vector of handles 276
12.1.4 Graphics object creation functions 277
12.1.5 Parenting 277
12.1.6 Positioning figures 278
12.2 Editing plots 279
12.2.1 Plot edit mode 279
12.2.2 Property Editor 280
12.3 Animation 281
12.3.1 Animation with Handle Graphics 282
12.4 Color etc. 285
12.4.1 Colormaps 285
12.4.2 Color of surface plots 287
12.4.3 Truecolor 288
12.5 Lighting and camera 288
12.6 Saving, printing and exporting graphs 289
12.6.1 Saving and opening figure files 289
12.6.2 Printing a graph 290
12.6.3 Exporting a graph 290

13 *Graphical User Interfaces (GUIs) 292
13.1 Basic structure of a GUI 292
13.2 A first example: getting the time 293
13.2.1 Exercise 297
13.3 Newton again 297
13.4 Axes on a GUI 301
13.5 Adding color to a button 302

PART II APPLICATIONS 305

14 Dynamical systems 307
14.1 Cantilever beam 309
14.2 Electric current 311
14.3 Free fall 314
14.4 Projectile with friction 323

15 Simulation 328
15.1 Random number generation 328
15.1.1 Seeding `rand` 329
15.2 Spinning coins 329
15.3 Rolling dice 330
15.4 Bacteria division 331
15.5 A random walk 331
15.6 Traffic flow 333
15.7 Normal (Gaussian) random numbers 336

16 *More matrices 341
16.1 Leslie matrices: population growth 341
16.2 Markov processes 345
16.2.1 A random walk 345
16.3 Linear equations 348
16.3.1 MATLAB's solution 349
16.3.2 The residual 350
16.3.3 Overdetermined systems 350
16.3.4 Underdetermined systems 351
16.3.5 Ill conditioning 351
16.3.6 Matrix division 352
16.4 Sparse matrices 354

17 *Introduction to numerical methods 359
17.1 Equations 359
17.1.1 Newton's method 359
17.1.2 The Bisection method 362

17.1.3 `fzero` 364
17.1.4 `roots` 364
17.2 Integration 364
17.2.1 The Trapezoidal rule 365
17.2.2 Simpson's rule 366
17.2.3 `quad` 367
17.3 Numerical differentiation 367
17.3.1 `diff` 368
17.4 First-order differential equations 369
17.4.1 Euler's method 369
17.4.2 Example: bacteria growth 370
17.4.3 Alternative subscript notation 371
17.4.4 A predictor-corrector method 373
17.5 Linear ordinary differential equations (LODEs) 374
17.6 Runge-Kutta methods 375
17.6.1 A single differential equation 375
17.6.2 Systems of differential equations: chaos 376
17.6.3 Passing additional parameters to an ODE solver 379
17.7 A partial differential equation 381
17.7.1 Heat conduction 381
17.8 Other numerical methods 385

Appendix A: Syntax quick reference 390
A.1 Expressions 390
A.2 Function M-files 390
A.3 Graphics 390
A.4 `if` and `switch` 391
A.5 `for` and `while` 392
A.6 Input/output 393
A.7 `load/save` 393
A.8 Vectors and matrices 393

Appendix B: Operators 395

Appendix C: Command and functionquick reference 396
C.1 General purpose commands 397
C.1.1 Managing commands 397
C.1.2 Managing variables and the workspace 397
C.1.3 Files and the operating system 397
C.1.4 Controlling the Command Window 398
C.1.5 Starting and quitting MATLAB 398
C.2 Logical functions 398

C.3 Language constructs and debugging 398
C.3.1 MATLAB as a programming language 398
C.3.2 Interactive input 399
C.4 Matrices and matrix manipulation 399
C.4.1 Elementary matrices 399
C.4.2 Special variables and constants 399
C.4.3 Time and date 400
C.4.4 Matrix manipulation 400
C.4.5 Specialized matrices 400
C.5 Mathematical functions 400
C.6 Matrix functions 401
C.7 Data analysis 402
C.8 Polynomial functions 402
C.9 Function functions 402
C.10 Sparse matrix functions 402
C.11 Character string functions 403
C.12 File I/O functions 403
C.13 Graphics 403
C.13.1 2-D 403
C.13.2 3-D 404
C.13.3 General 404

Appendix D: ASCII character codes **405**

Appendix E: Solutions to selected exercises **406**

Index **421**

This page intentionally left blank

In memory of Brian Hahn 1946–2005

Daniel T. Valentine and the staff at Butterworth-Heinemann, Elsevier would like to dedicate this book to the memory of Brian Hahn, who wrote the first edition in 1997 while he was an Associate Professor in the Department of Mathematics and Applied Mathematics, University of Cape Town, South Africa. Brian's academic career began after a PhD in Theoretical Physics obtained from Cambridge. His first post as Lecturer was at the University of the Witwaterstrand, Johannesburg. He was promoted to Senior Lecturer in 1979. He joined the University of Cape Town in the same year and was appointed Associate Professor in Applied Mathematics in 1991. Brian served as Head of Department for five years. He was a loved and respected teacher whose expertise in modeling and computing translated so well into the first year Applied Mathematics courses. He was the author of more than ten books on programming languages. We trust that the third edition of this book will continue to help students understand and exploit the full power of MATLAB both as a mathematical tool and as a programming language.

This page intentionally left blank

Preface to the third edition

The main purpose in planning a third edition of this book was to upgrade it to cover the latest version of MATLAB (Version 7.2 Release 2006a).

The other purpose was to maintain the objectives of the late Brian D. Hahn as stated in the prefaces of the first and second edition. In his prefaces to the first and second edition, he pointed out the following:

This book presents the MATLAB computer programming system as a problem-solving tool for scientists and engineers who have no prior knowledge of computer programming. It is based on a teach-yourself approach; readers are frequently invited to experiment for themselves in order to discover how particular constructs work. The text was originally written as a companion to a hands-on course at the University of Cape Town. Most of the students taking the course had little or no computing experience and came from disadvantaged backgrounds. Consequently the book assumes that readers have no knowledge of computing, unlike most similar books on MATLAB.

MATLAB is based on the mathematical concept of a matrix. Again, unlike most comparable books, this text does not assume any knowledge of matrices on the part of the reader; in fact the concept is developed gradually, as the context requires it. Since the book is written primarily for scientists and engineers, some of the examples of necessity involve some first-year university mathematics, particularly in the last chapter. However, these examples are self-contained, and omitting them will not detract from the development of your programming skills.

MATLAB can be used in two distinct modes. In keeping with the present age's craving for instant gratification, it offers immediate execution of statements, or even groups of statements, in the Command Window. For the more patient, it also offers conventional programming by means of script files. This book makes good use of both modes. On the one hand, it encourages the use of cut-and-paste techniques to take full advantage of the interactive Windows environment,

while on the other hand also stressing programming principles and algorithm development, with the help of structure plans.

Although most of MATLAB's basic features are covered, the book is neither an exhaustive nor systematic reference manual, since this would not be in keeping with its informal style. Constructs, such as `for` *and* `if`*, are not therefore always introduced in their most general form initially, as is common in many texts, but rather more gradually in the most natural places throughout the book. On the other hand, many texts present these constructs somewhat superficially; this book attempts to discuss them thoroughly. For the curious, there are helpful syntax and function quick references in the appendices.*

MATLAB by its nature lends itself to a number of pitfalls for the unwary beginner. The text warns the user of these wherever possible.

The fundamentals of MATLAB are motivated throughout with many examples, from a number of different scientific and engineering areas, such as simulation, population modeling, and numerical methods, as well as from business and everyday life. Beginners, as well as experienced programmers wishing to learn MATLAB as an additional language, will therefore find plenty of interest in the book.

Emphasis is also placed on programming style throughout the book—writing clear and readable code.

Each chapter concludes with a summary of the MATLAB features introduced in the chapter.

There is a large collection of exercises at the end of each chapter, gleaned from the author's many years' experience of running hands-on programming courses for beginners and professionals alike, in BASIC, Pascal, C, C++ and MATLAB. Complete solutions to many of the exercises appear in an appendix.

There is a comprehensive and instructive index.

For the second edition, in working my way through Version 6, I found so many interesting new features (for example, GUIs) that I was unable to resist incorporating most of them into the text. Consequently I decided to split the book into two parts. Part I contains what I consider to be the real essentials; Part II has everything else.

In this edition I have attempted to retain the style and approach of the first edition: informal, aimed at beginners, and with plenty of examples from science and engineering. Several of the chapters from the previous editions, which

feature the essential elements of MATLAB, have been brought together to form Part 1. In addition, I have added two new chapters. These are Chapters 3 and 14.

Chapter 3 describes a structured step-by-step method to achieve top-down design and algorithm development. The steps in the design process are applied in several examples. The intention is to get students thinking about how they need to formulate a problem to successfully utilize MATLAB. Chapter 14 on **Dynamical Systems** provides straightforward applications of the tools described and examined in the first 10 chapters. The problems solve are on relatively simple dynamical systems of engineering and scientific interest. Since this book is an introductory course on MATLAB, a tool for technical computing, the examples are mathematical formulations of problems from first courses in science and engineering. The purpose of the text is to provide instruction on how to solve the mathematical problems needed to gain insight into science and engineering. Thus, these eleven chapters (skipping over the sections marked with an asterisk) are sufficient for a first course in MATLAB. (For the computer and programming-calculator wise students, the chapters on more advanced topics should help them get into the application of MATLAB to solve the more complex problems confronted in upper division courses at university and, subsequently, on the job.)

This book can be used as a course textbook or for student self-study. For the latter, it is a useful supplemental text for any course in science and engineering. The instructor, of course, provides the necessary encouragement, enthusiasm and guidance to help the student begin to learn the power of MATLAB to solve numerous problems that engineers and scientists formulate in terms of mathematics and, hence, help the student begin to master MATLAB. The book is written as a sequence of exercises, and the reader would benefit from doing the exercises within the text as well as doing some of the exercises at the end of the chapters.

To the student: I recommend that you read the text while you are at your computer so that you can do the exercises with MATLAB. It will be useful and fun for you to go through the exercises with the purpose of discovering how MATLAB does what it is commanded by you to do. *You learn how to use a tool like MATLAB through hands-on experience.* This, of course, is a good thing because it is quite pleasurable to learn by doing and, hence, discover how to use MATLAB to enhance your learning of engineering and science by tapping the wealth of capability at your disposal in MATLAB. You will discover immediately that computer tools produce correct answers only when commands and input data are accurate and correct (no typographical errors are tolerated). 'Debugging'—finding the errors in your typed command lines—is a big part of the game that is played

when you create computer programs to solve your technical problems. Going from the development of a structured-plan to the translation of your plan into a series of commands in MATLAB, debugging and, ultimately getting answers is very rewarding as you will discover. Enjoy!

On-line supplements: In addition to the material covered in the text, for the instructors who adopt the text and for students who purchase the text, there is a website that provides a set of examples on a variety of topics as well as exercises, further problems and Powerpoint slides to help instructors, students and self-learners discover the wealth of capabilities of MATLAB. Go to www.textbooks.elsevier.com for more information.

I wish to acknowledge the support of Mary and Clara and dedicate this book to them.

Daniel T. Valentine
clara@clarkson.edu
December 2006

Part I

Essentials

Part I is concerned with those aspects of MATLAB which you need to know in order to get to grips with the essentials of MATLAB and of technical computing. Chapters 11, 12 and 13 are marked with an asterisk, *. Some sections in other chapters are similarly marked. These sections and chapters can be skipped in your first reading and step-up-step execution of all of the MATLAB commands and scripts described in your book. This book is a tutorial and, hence, you are expected to use MATLAB extensively while you go through the text.

This page intentionally left blank

1 Introduction

The objectives of this chapter are to enable you to use some simple MATLAB commands from the Command Window and to examine various MATLAB desktop and editing features.

MATLAB is a powerful computing system for handling the calculations involved in scientific and engineering problems. The name MATLAB stands for MATrix LABoratory, because the system was designed to make matrix computations particularly easy. If you don't know what a matrix is, don't worry—we will look at them in detail later.

This book assumes that you have never used a computer before to do the sort of scientific calculations that MATLAB handles. You will, however, need to be able to find your way around a computer keyboard and the operating system running on your computer (e.g. Windows or UNIX). The only other computer-related skill you will need is some very basic text editing.

One of the many things you will like about MATLAB (and which distinguishes it from many other computer programming systems, such as C++ and Java) is that you can use it *interactively*. This means you type some commands at the special MATLAB prompt, and get the answers immediately. The problems solved in this way can be very simple, like finding a square root, or they can be much more complicated, like finding the solution to a system of differential equations. For many technical problems you have to enter only one or two commands, and you get the answers at once. MATLAB does most of the work for you.

There are two essential requirements for successful MATLAB programming:

- You need to learn the *exact* rules for writing MATLAB statements.
- You need to develop a logical plan of attack for solving particular problems.

This chapter is devoted mainly with an introduction to the first requirement: learning some basic MATLAB rules. Computer programming is a precise science

(some would say it is also an art); you have to enter statements in *precisely* the right way. If you don't, you will get rubbish. There is a saying among computer programmers:

Garbage in, garbage out.

If you give MATLAB a garbage instruction, you will get a garbage result. With experience, you will see that with MATLAB you can design, develop and implement computational and graphical tools to do relatively complex science and engineering problems. Of course, some of the necessary experience required to tap the full potential of MATLAB is your continuing education both before and after graduation. With MATLAB you will be able to adjust the look, modify the way you interact with MATLAB, and develop a toolbox of your own that helps you solve problems that are of interest to you. In other words, you can, with significant experience, customize your MATLAB working environment.

As you learn the basics of MATLAB and, for that matter, any other computer tool, remember that computer applications do nothing randomly. Hence, as you use MATLAB, observe and study all responses from the command-line operations that you implement. You need to learn what this tool does and what it doesn't do. To begin an investigation into the capabilities of MATLAB, we will do relatively simple problems. Initially, we must do problems to which we know the answers because we are evaluating the tool and its capabilities. This is the first step in the process of learning how to use any tool. As you learn about MATLAB, you are also going to learn about computer programming. You need to do this for two reasons: (1) To create your own computational tools. (2) To appreciate the difficulties involved in the design of efficient, robust and accurate computational and graphical tools (i.e. computer programs).

In the rest of this chapter we will look at some simple examples for you to try out. Don't bother about understanding exactly what is happening. The understanding will come in later chapters when we look at the details. It is very important for you to exercise the MATLAB tool to learn by hands-on experience how it works. Once you have examined a few of the basic rules in this chapter, you will be prepared to master many of the basic rules presented in the next chapter. This will help you go on to solve more interesting and substantial problems. Finally, in the last section of this chapter you will take a quick tour of the MATLAB desktop.

1.1 Using MATLAB

In order to use MATLAB it must either be installed on your computer, or you must have access to a network where it is available. Throughout this book

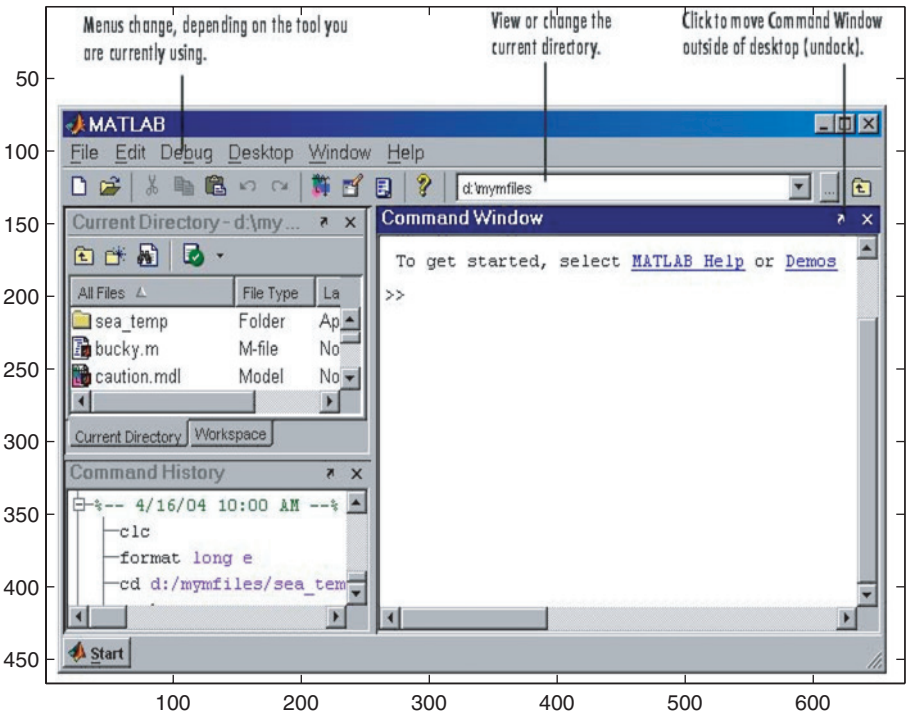


Figure 1.1 The MATLAB desktop

the latest version of MATLAB at the time of writing is assumed—Version 7.2 (Release 2006a).

To start MATLAB from Windows, double-click the MATLAB icon on your Windows desktop. To start it from a UNIX platform, type `matlab` at the operating system prompt. When MATLAB starts, the MATLAB desktop opens as shown in Figure 1.1. The window in the desktop that concerns us for this chapter is the Command Window, where the special ≫ prompt appears. This prompt means that MATLAB is waiting for a command. You can quit MATLAB at any time with one of the following:

- Select **Exit MATLAB** from the desktop **File** menu.
- Enter `quit` or `exit` at the Command Window prompt.

Do *not* click on the close box in the top right corner of the MATLAB desktop. This does not allow MATLAB to terminate properly and, on rare occasions, may cause problems with your computer operating software (the new author corrupted a graphics utility when doing color graphics by clicking the red x!).

Once you have started MATLAB, try the following exercises in the Command Window. If necessary, make the Command Window active by clicking anywhere inside its border.

1. Since we have experience doing arithmetic, we want to examine if MATLAB does this correctly. When applying computer tools this is a required step to gain confidence in the tool and in our ability to use the tool.

(a) Type `2+3` after the ≫ prompt, followed by **Enter**, i.e. press the **Enter** key, as indicated by <**Enter**>, below:

```
≫ 2+3    <Enter>
```

Commands are only carried out when you press **Enter**. The answer in this case is, of course, 5.

(b) Next try the following:

```
≫ 3-2    <Enter>

≫ 2*3    <Enter>

≫ 1/2    <Enter>

≫ 2^3    <Enter>

≫ 2\1    <Enter>
```

What about `(1)/(2)` and `(2)^(3)`? Can you figure out what the symbols `*`, `/` and `^` mean? Yes, they are multiplication, division and exponentiation. The backslash means the denominator is to the left of the symbol and the numerator is to the right of the symbol; the result for the last command is 0.5. This operation is equivalent to `1/2`.

(c) Try the following commands:

```
≫ 2 .* 3    <Enter>

≫ 1 ./ 2    <Enter>

≫ 2 .^ 3    <Enter>
```

In the three examples a dot (full stop, period) in front of the `*`, `/` and `^`, respectively, does not change the results in these examples because the multiplication, division, and exponentiation is done with

single numbers—an explanation for the need for these symbols is provided later when we deal with arrays of numbers!

The following items are hints on creating and editing command lines:

- The line with the ≫ prompt is called the *command line*.
- You can edit a MATLAB command before pressing **Enter** by using various combinations of the **Backspace**, **Left-arrow**, **Right-arrow** and **Del** keys. This helpful feature is called *command line editing*.
- You can select (and edit) previous commands you have entered using **Up-arrow** and **Down-arrow**. But remember to press **Enter** to get the command carried out (i.e. to *run* or to *execute* the command).
- MATLAB has a useful editing feature called *smart recall*. Just type the first few characters of the command you want to recall, e.g. type the characters `2*` and press the **Up-arrow** key—this recalls the most recent command starting with `2*`.

(d) How do you think MATLAB would handle `0/1` and `1/0`? Try it. MATLAB is sensible about anticipating some errors; it warns you in case you didn't realize you were dividing by zero, but still gives the answer `Inf`. If you insist on using ∞ in a calculation, which you may legitimately wish to do, type the symbol `Inf` (short for *infinity*), e.g. try `13+Inf` and `29/Inf`.

(e) Another special value that you may meet is `NaN`, which stands for *Not-a-Number*. It is the answer to calculations like `0/0`.

2. Now let us assign values to variables to do arithmetical operations with the variables.

(a) Enter the command (in programming jargon a *statement*) `a = 2`, i.e. the MATLAB command line should look like this:

```
≫ a = 2    <Enter>
```

The symbol `a` is called a *variable*. This statement *assigns* the value of 2 to `a`. (Note that this value is displayed immediately after the statement is executed.) Now try entering the statement `a = a + 7` followed on a new line by `a = a * 10`. Do you agree with the final value of `a`? Do we agree that it is 90?

(b) Now enter the statement

```
≫ b = 3;    <Enter>
```

The semicolon (`;`) prevents the value of `b` from being displayed. However, `b` still has the value 3 as you can see by entering its name

without a semicolon, i.e. by executing the following command: ≫ `b` <**Enter**>.

(c) Assign any values you like to two variables `x` and `y`. Now see if you can in a single statement assign the *sum* of `x` and `y` to a third variable `z`. One way of doing this is

```
≫ x = 2; y = 3;  <Enter>
≫ z = x + y  <Enter>
```

Notice that, in addition to doing the arithmetic with variables with assigned values, several commands separated by semicolons (or by commas) can be put on one line.

3. MATLAB has all of the usual mathematical functions that are on a scientific-electronic calculator, like `sin`, `cos`, `log` (meaning the *natural* logarithm), as well as a lot more; see Appendix C.5 for many more examples:

(a) Find $\sqrt{\pi}$ with the command `sqrt(pi)`. The answer should be 1.7725. Note that MATLAB knows the value of `pi`, because it is one of MATLAB's many built-in functions.

(b) Trigonometric functions like `sin(x)` expect the argument `x` to be in *radians*. Multiply degrees by $\pi/180$ to get radians. For example, use MATLAB to calculate $\sin(90°)$. The answer should be 1, i.e. `sin(90*pi/180)`.

(c) The exponential function e^x is computed in MATLAB as `exp(x)`. Use this information to find e and $1/e$ (2.7183 and 0.3679).

Note that, because of the numerous built-in functions like `pi` or `sin`, care must be exercised in the naming of user-defined variables. The name of user-defined variables should not duplicate the name of a built-in function unless it is deliberately done for a good reason. This problem can be illustrated as follows. Try the following:

```
≫ pi = 4  <Enter>

≫ sqrt(pi)  <Enter>

≫ whos  <Enter>

≫ clear pi  <Enter>

≫ whos  <Enter>

≫ sqrt(pi)  <Enter>
```

≫ `clear` <**Enter**>

≫ `whos` <**Enter**>

Note that `clear` executed by itself clears all of the local variables in the *workspace*. Executing the command ≫ `clear pi` clears the locally defined variable `pi`. Hence, if you decided to redefine a built-in function or command, the newly defined value is used! The command `whos` is executed to determine the list of local variables or commands presently in the workspace. The first execution of the command `pi = 4` in the above example displays your redefinition of the built-in number `pi`. It is a 1-by-1 (or `1x1`) *double array*, which means this *data type* was created when `pi` was assigned a number (you will learn more about other data types later, as we proceed in our investigation of MATLAB).

4. MATLAB has numerous general functions. For example, try `date` and `calendar` for starters.

5. MATLAB also has numerous *commands*, such as `clc` (for *clear command window*). `help` is another command you will use a lot (see below). The difference between functions and commands is that functions usually return with a value, e.g. the date, while commands tend to change the environment in some way, e.g. by clearing the screen, or saving some statements to the workspace (on the disk).

6. Variables such as `a` and `b` above are called *scalars*; they are single-valued. MATLAB also handles *vectors* (generally referred to in MATLAB as *arrays*), which are the key to many powerful features of the language. The easiest way of defining a vector where the elements (components) increase by the same amount is with a statement like

   ≫ `x = 0 : 10;` <**Enter**>

   That is a *colon* (`:`) between the 0 and the 10. There's no need to leave a space on either side of it, but it makes it more readable. Enter `x` to check that `x` is a vector; it is a *row vector*, i.e. it is a single row by 11 columns array. Type the following command to verify that this is the case:

   ≫ `size(x)` <**Enter**>.

   As a further introduction to *vectors* continue this exercise by examining the following examples.

   (a) Part of the real power of MATLAB is illustrated by the fact that other vectors can now be defined (or created) in terms of the vector `x` just defined. Try

   ≫ `y = 2 .* x` <**Enter**>

   ≫ `w = y ./ x` <**Enter**>

and

```
≫ z = sin(x)   <Enter>
```

(no semicolons). Note that the first command line created a vector `y` by multiplying each element of `x` by the factor `2`. The second command line is an array operation. It created a vector `w` by taking each element of `y` and dividing it by the corresponding element of `x`. Since each element of `y` is two times the corresponding element of `x`, the vector `w` is a row vector of 11 elements all equal to `2`. Finally, `z` is a vector with the `sin(x)` as its elements.

(b) All you have to do to draw a reasonably nice graph of sin(x) is to enter the following commands:

```
≫ x = 0 : 0.1 : 10;  <Enter>

≫ z = sin(x);  <Enter>

≫ plot(x,z), grid  <Enter>
```

The graph appears in a separate figure window (see Figure 1.2). You can select the Command Window or figure windows by clicking anywhere inside them. The **Windows** pull-down menus can be used in any of these windows.

Note that the first command line above has three numbers after the equal sign. When there are three numbers separated by two colons in this way, the middle number is the *increment*. The increment of `0.1` was selected to give a reasonably smooth graph. The command `grid` following the comma in the last command line adds a grid to the graph.

(c) If you want to see more cycles of the sine graph just use command-line editing to change `sin(x)` to `sin(2*x)`.

(d) Try drawing the graph of `tan(x)` over the same domain. You may find aspects of your graph surprising. A more accurate version is presented in Chapter 5. An alternative way to examine mathematical functions graphically is to use the following command:

```
≫ ezplot('tan(x)')  <Enter>.
```

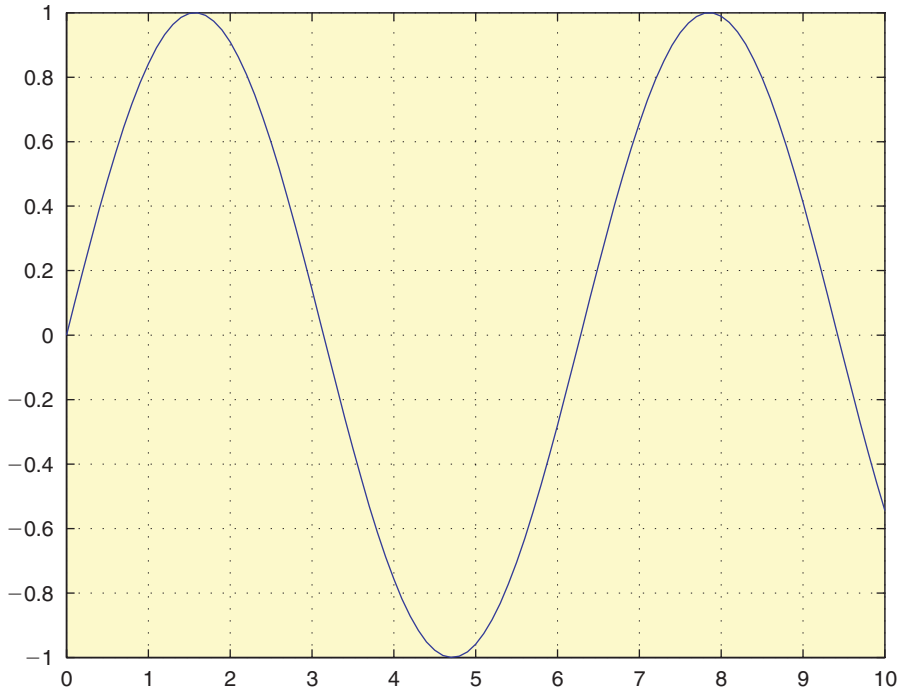


Figure 1.2 A figure window

The apostrophes around the function `tan(x)` are important in the `ezplot` command; also note that the default domain of `x` in `ezplot` is *not* 0 to 10.

(e) Another useful Command Window editing feature is *tab completion*: type the first few letters of a MATLAB name and then press the **Tab** key. If the name is unique, it is automatically completed. If the name is not unique, press the **Tab** a second time to see all the possibilities. Try this feature out, e.g. by typing `ta` at the command line followed by **Tab** twice.

7. Systems of linear equations are very important in engineering and scientific analysis. A simple example is finding the solution of two simultaneous equations, e.g.

$$\begin{aligned} x + 2y &= 4, \\ 2x - y &= 3. \end{aligned}$$

Two approaches to the solution of this system of equations are given next. This is followed by a check of the results using the arithmetic operations you have already learned to used.

(a) Approach 1: This approach is the *matrix* method approach. Type the following commands (exactly as they are):

```
≫ a = [1 2; 2 -1];  <Enter>
≫ b = [4; 3];  <Enter>
≫ x = a\b  <Enter>
```

which result in

```
x =
     2
     1
```

i.e. $x = 2$, $y = 1$.

(b) Approach 2: This approach uses the built-in `solve` function. Type the following commands (exactly as they are):

```
≫ [x,y] = solve('x+2*y=4','2*x-y=3')  <Enter>
```

```
≫ whos  <Enter>
≫ x = double(x), y = double(y)  <Enter>
≫ whos  <Enter>
```

The function *double* converts `x` and `y` from symbolic objects (another *data type* in MATLAB) to double arrays (i.e. the numerical-variable data type associated with an assigned number).

(c) Check of results: After executing either (a) or (b) above type the following commands (exactly as they are):

```
≫ x + 2*y % should give ans = 4  <Enter>
≫ 2*x - y % should give ans = 3  <Enter>
```

The `%` symbol is a *flag* that indicates all information to the right is *not* part of the command. It is a comment. (We will examine the need for comments when we learn to develop coded programs of command lines later on in the text.)

8. If you want a spectacular sample of what MATLAB has to offer, try `demo` at the command line.

Alternatively, double-click **Demos** in the Launch Pad, which is found by clicking the START button in the lower left-hand corner of the MATLAB desktop. (If you can't see **Demos**, click on the question mark to open the help

browser, or you can launch the demonstration programs by clicking on it in `Demos` in the pull-down menu under `Help` at the top of the MATLAB desktop.) For a listing of demonstration programs by category try `help demos`.

9. MATLAB has a very useful 'help' system, which we look at in a little more detail in the last section of this chapter. For the moment type `help` at the command line to see all the categories on which you can get help. For example, type `help elfun` to see all MATLAB's elementary mathematical functions. Another utility is `lookfor`; it enables you to search for a particular string in the `help` text of functions, e.g. `lookfor eigenvalue` displays all the functions relating to eigenvalues. (There is one problem with the results that you get in the command window. The commands are, for emphasis only, in upper case; when used they must be typed as lower case. This is because the latest versions of MATLAB are case sensitive; hence, `a` and `A` are considered different names. In light of this fact, it is safer and more instructive to use the help manuals by clicking the question mark (?) in the task bar at the top of the MATLAB desktop window. The examples are correctly reproduced in the help manuals found by clicking on the `?` in one of the MATLAB toolbars near the top of the MATLAB desktop window.)

10. MATLAB has all sorts of other goodies. For example, you can generate a 10-by-10 (or 10×10) *magic square* by executing the command `magic(10)`, where the rows, columns and the main diagonal all add up to the same value. Try it. In general, an $n \times n$ magic square has a row and column sum of $n(n^2 + 1)/2$.

You can even get a *contour* plot of the elements of a magic square. MATLAB pretends that the entries in the square are heights above sea level of points on a map, and draws the contour lines. `contour(magic(22))` looks rather nice.

11. If you want to see the famous Mexican hat shown in Figure 1.3, enter the following four lines (be careful not to make any typing errors):

```
≫ [x y ] = meshgrid(-8 : 0.5 : 8);  <Enter>
≫ r = sqrt(x.^2 + y.^2) + eps;  <Enter>
≫ z = sin(r) ./ r;  <Enter>
≫ mesh(z);  <Enter>
```

Try `surf(z)` to generate a faceted (tiled) view of the surface.

`surfc(z)` or `meshc(z)` draws a 2-D contour plot under the surface.

The command

```
≫surf(z), shading flat  <Enter>
```

produces a rather nice picture by removing the grid lines.

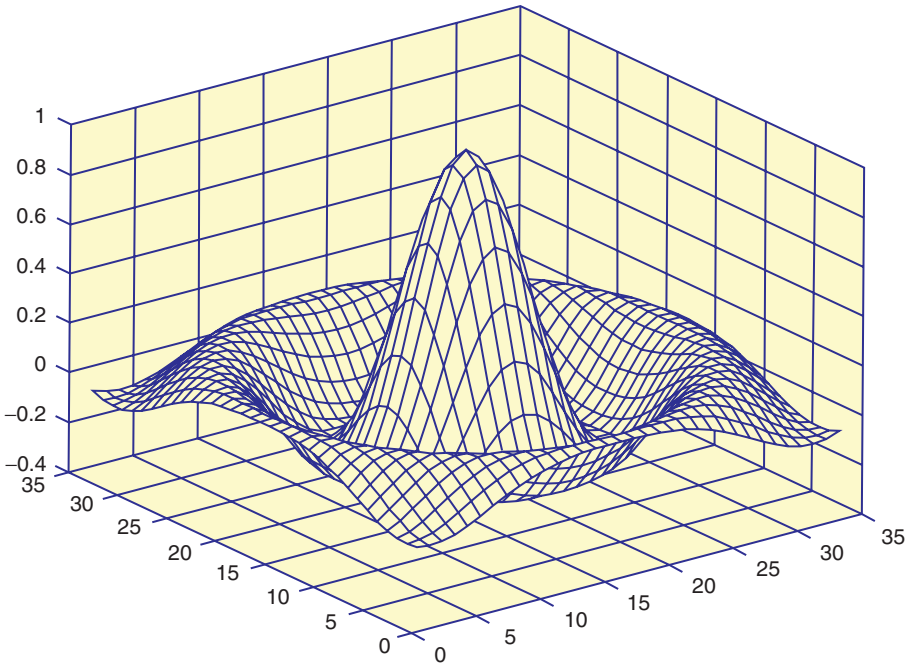


Figure 1.3 The Mexican hat

12. If your PC has a speaker you could try

```
≫load handel  <Enter>
≫sound(y,Fs)  <Enter>
```

for a snatch of Handel's Hallelujah Chorus.

For some different sounds you can try loading `chirp`, `gong`, `laughter`, `splat` and `train`. You have to run `sound(y,Fs)` for each one.

13. If you want to see a view of the Earth from space, try:

```
≫load earth  <Enter>
≫image(X); colormap(map)  <Enter>
≫axis image  <Enter>
```

14. If you are really bored, try `why`. Why not? Next try `why(2)` twice. Finally, to see the MATLAB code that does this, type the following command:

```
≫edit why  <Enter>
```

Once you have looked at this file close it via the pull-down menu by clicking `File` at the top of the Editor desktop window and then `Close Editor`; do *not save* it, in case you accidently typed something and modified the

file. The `edit` command will be used soon to illustrate the creation of an *M-file* like `why.m`, which is the name of the file executed by the command `why`. You will create an *M-file* after we go over some of the basic features of the MATLAB desktop—more details on creating programs in the MATLAB environment will be explained when the editor is introduced in more detail in Chapter 2.

1.2 The MATLAB desktop

The default MATLAB desktop is shown in Figure 1.1. It should immediately be stressed that MATLAB has an extremely useful online Help system. If you are serious about learning MATLAB you should make it your business to work through the various Help features, starting with the section on the desktop.

To get into the Help browser, either click on the help button (`?`) in the desktop toolbar, or select the **Help** menu in any tool. Having opened the Help browser, select the **Contents** tab in the **Help Navigator** pane on the left. To get to the desktop section expand successively the MATLAB, **Getting Started** and **Development Environment** items. **MATLAB Desktop** is listed under the latter. When you've looked at the **Desktop** item you should go on to **Desktop Tools**. (If you mess up your desktop and want to get back to the default, select the **View** -> **Desktop Layout** -> **Default** menu item.)

The desktop contains a number of *tools*. We have already used the Command Window. On the left is the Current Directory, which shares a 'docking' position with the Workspace browser. Use the tabs to switch between the Current Directory and the Workspace browser. Below the Current Directory you will find the Command History.

You can resize any of these windows in the usual way. A window can be moved out of the MATLAB desktop by *undocking* it. Do this either by clicking on the arrow in the window's title bar, or by making the window active (click anywhere inside it) and then selecting **Undock** from the **View** menu.

To dock a tool window that is outside the MATLAB desktop (i.e. to move it back into the desktop) select **Dock** from its **View** menu.

You can group desktop windows so that they occupy the same space in the MATLAB desktop. Access to the individual windows is then by means of their tabs. To group windows in this way drag the title bar of one window on top of the title bar

of the other window. The outline of the window you're dragging overlays the target window, and the bottom of the outline includes a tab. The Status bar informs you to `Release the mouse button to tab-dock these windows`. If you release the mouse the windows are duly tab-docked in the outlined position.

There are six predefined MATLAB desktop configurations, which you can select from the **View** -> **Desktop Layout** menu.

1.3 Sample program

In Section 1.1 we saw some simple examples of how to use MATLAB by entering single commands or statements at the MATLAB prompt. However, you might want to solve problems which MATLAB can't do in one line, like finding the roots of a quadratic equation (and taking all the special cases into account). A *collection* of statements to solve such a problem is called a *program*. In this section we look at the mechanics of writing and running two short programs, without bothering too much about how they work—explanations will follow in the next chapter.

1.3.1 Cut and paste

Suppose you want to draw the graph of $e^{-0.2x}\sin(x)$ over the domain 0 to 6π, as shown in Figure 1.4. The Windows environment lends itself to nifty *cut and paste* editing, which you would do well to master. Proceed as follows.

From the MATLAB desktop select **File** -> **New** -> **M-file**, or click the new file button on the desktop toolbar (you could also type `edit` in the Command Window followed by Enter). This action opens an Untitled window in the Editor/Debugger. You can regard this for the time being as a 'scratch pad' in which to write programs. Now type the following two lines in the Editor, exactly as they appear here:

```
x = 0 : pi/20 : 6 * pi;
plot(x, exp(-0.2*x) .* sin(x), 'r'),grid
```

Incidentally, that *is* a dot (full stop, period) in front of the second `*` in the second line—a more detailed explanation later! The additional argument `'r'` for `plot` will draw a red graph, just to be different.

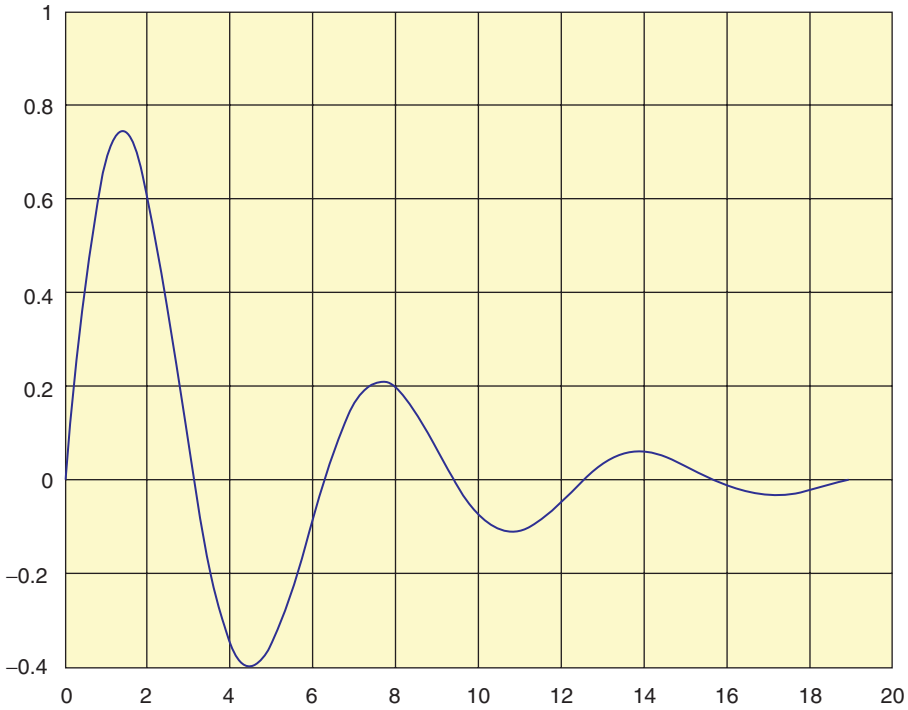


Figure 1.4 $e^{-0.2x} \sin(x)$

Next, move the mouse pointer (which now looks like a very thin capital I) to the left of the `x` in the first line. Keep the left mouse button down while moving the mouse pointer to the end of the second line. This process is called *dragging*. Both lines should be highlighted at this stage, probably in blue, to indicate that they have been *selected*.

Select the **Edit** menu in the Editor window, and click on **Copy** (or just use the keyboard shortcut **Ctrl+C**). This action copies the highlighted text to the Windows *clipboard*, assuming that your operating system is Windows.

Now go back to the Command Window. Make sure the cursor is positioned at the `>>` prompt (click there if necessary). Select the **Edit** menu, and click on **Paste** (or use the **Ctrl+V** shortcut). The contents of the clipboard will be copied into the Command Window. To execute the two lines in the program, press **Enter**. The graph should appear in a figure window.

This process, from highlighting (selecting) text in the Editor, to copying it into the Command Window, is called 'cut and paste' (more correctly 'copy and paste' here, since the original text is copied from the Editor, rather than being cut from it). It's well worth practicing until you have it right.

If you need to correct the program, go back to the Editor, click at the position of the error (this moves the *insertion point* to the right place), make the correction, and cut and paste again. Alternatively, you can use command-line editing to correct mistakes. As yet another alternative, you can paste from the Command History window (which incidentally goes back over many previous sessions). To select multiple lines in the Command History window keep **Ctrl** down while you click.

If you prefer, you can enter multiple lines directly in the Command Window. To prevent the whole group from running until you have entered the last line use **Shift+Enter** or **Ctrl+Enter** after each line until the last. Then press **Enter** to run all the lines.

As another example, suppose you have $1000 saved in the bank. Interest is compounded at the rate of 9 percent per year. What will your bank balance be after one year? Now, if you want to write a MATLAB program to find your new balance, you must be able to do the problem yourself in principle. Even with a relatively simple problem like this, it often helps first to write down a rough *structure plan*:

1. Get the data (initial balance and interest rate) into MATLAB.
2. Calculate the interest (9 percent of $1000, i.e. $90).
3. Add the interest to the balance ($90 + $1000, i.e. $1090).
4. Display the new balance.

Go back to the Editor. To clear out any previous text, select it as usual by dragging (or use **Ctrl+A**), and press the **Del** key. By the way, to deselect highlighted text, click anywhere outside the selection area. Enter the following program, and then cut and paste it to the Command Window:

```
balance = 1000;
rate = 0.09;
interest = rate * balance;
balance = balance + interest;
disp( 'New balance:' );
disp( balance );
```

When you press **Enter** to run it, you should get the following output in the Command Window:

```
New balance:
            1090
```

1.3.2 Saving a program: script files

We have seen how to cut and paste between the Editor and the Command Window in order to write and run MATLAB programs. Obviously you need to save the program if you want to use it again later.

To save the contents of the Editor, select **File -> Save** from the Editor menubar. A **Save file as:** dialogue box appears. Select a directory and enter a file name, which *must* have the extension `.m`, in the **File name:** box, e.g. `junk.m`. Click on **Save**. The Editor window now has the title `junk.m`. If you make subsequent changes to `junk.m` in the Editor, an asterisk appears next to its name at the top of the Editor until you save the changes.

A MATLAB program saved from the Editor (or any ASCII text editor) with the extension `.m` is called a *script file*, or simply a *script*. (MATLAB *function* files also have the extension `.m`. MATLAB therefore refers to both script and function files generally as *M-files*.) The special significance of a script file is that, if you enter its name at the command-line prompt, MATLAB carries out each statement in the script file as if it were entered at the prompt.

The rules for script file names are the same as those for MATLAB variable names (see the next Chapter 2, Section 2.1).

As an example, save the compound interest program above in a script file under the name `compint.m`. Then simply enter the name

```
compint
```

at the prompt in the Command Window (as soon as you hit Enter). The statements in `compint.m` will be carried out exactly as if you had pasted them into the Command Window. You have effectively created a new MATLAB command, viz., `compint`.

A script file may be listed in the Command Window with the command `type`, e.g.

```
type compint
```

(the extension `.m` may be omitted).

Script files provide a useful way of managing large programs which you do not necessarily want to paste into the Command Window every time you run them.

Current directory

When you run a script, you have to make sure that MATLAB's current directory (indicated in the **Current Directory** field on the right of the desktop toolbar) is set to the directory in which the script is saved. To change the current directory type the path for the new current directory in the **Current Directory** field, or select a directory from the drop-down list of previous working directories, or click on the browse button (`...`) to select a new directory.

The current directory may also be changed in this way from the Current Directory browser in the desktop.

You can change the current directory from the command line with `cd` command, e.g.

```
cd \mystuff
```

The command `cd` by itself returns the name of the current directory.

Recommended way to run a script from the Current Directory browser

A handy way to run a script is as follows. Select the file in the Current Directory browser. Right-click it. The *context menu* appears (context menus are a general feature of the desktop). Select **Run** from the context menu. The results appear in the Command Window. If you want to edit the script, select **Open** from the context menu.

To specify the types of files shown in the Current Directory Browser, use **View -> Current Directory Filter**. For example, you may wish to see only M-files. Alternatively, you can right-click the column title in the Current Directory browser and use the context menu.

You can sort the files in the Current Directory browser by clicking the title of the file column. A click 'toggles' the sort, i.e. from ascending to descending order or vice versa.

The Help portion of an M-file appears in the pane at the bottom of the Current Directory browser. Try this with an M-file in one of the MATLAB directories, e.g. `\toolbox\matlab\elmat`. A one-line description of the M-file appears in the right-most column of the Current Directory browser (you may need to enlarge the browser to see all the columns).

1.3.3 How a program works

We will now discuss in detail how the compound interest program works.

The MATLAB system is technically called an *interpreter* (as opposed to a *compiler*). This means that each statement presented to the command line is translated (interpreted) into language the computer understands better, and then *immediately* carried out.

A fundamental concept in MATLAB is how numbers are stored in the computer's *random access memory* (RAM). If a MATLAB statement needs to store a number, space in the RAM is set aside for it. You can think of this part of the memory as a bank of boxes or *memory locations*, each of which can hold only one number at a time. These memory locations are referred to by symbolic names in MATLAB statements. So the statement

```
balance = 1000
```

allocates the number 1000 to the memory location named `balance`. Since the contents of `balance` may be changed during a session it is called a *variable*.

The statements in our program are therefore interpreted by MATLAB as follows:

1. Put the number 1000 into variable `balance`.
2. Put the number 0.09 into variable `rate`.
3. Multiply the contents of `rate` by the contents of `balance` and put the answer in `interest`.
4. Add the contents of `balance` to the contents of `interest` and put the answer in `balance`.
5. Display (in the Command Window) the message given in single quotes.
6. Display the contents of `balance`.

It hardly seems necessary to stress this, but these interpreted statements are carried out *in order from the top down*. When the program has finished running, the variables used will have the following values:

```
balance  : 1090
interest : 90
rate     : 0.09
```

Note that the original value of `balance` (1000) is lost.

Try the following exercises:

1. Run the program as it stands.
2. Change the first statement in the program to read

   ```
   balance = 2000;
   ```

   Make sure that you understand what happens now when the program runs.
3. Leave out the line

   ```
   balance = balance + interest;
   ```

   and rerun. Can you explain what happens?
4. Try to rewrite the program so that the original value of `balance` is *not* lost.

A number of questions have probably occurred to you by now, such as

- What names may be used for variables?
- How can numbers be represented?
- What happens if a statement won't fit on one line?
- How can we organize the output more neatly?

These questions will be answered in the next chapter. However, before we write any more complete programs there are some additional basic concepts which need to be introduced. These concepts are introduced in the next chapter.

Summary

- MATLAB is a matrix-based computer system designed to assist in scientific and engineering problem solving.
- To use MATLAB, you enter commands and statements on the command line in the Command Window. They are carried out immediately.
- `quit` or `exit` terminates MATLAB.
- `clc` clears the Command Window.
- `help` and `lookfor` provide help.

- `plot` draws an *x*–*y* graph in a figure window.
- `grid` draws grid lines on a graph.

EXERCISE

1.1 Give values to variables `a` and `b` on the command line, e.g. `a = 3` and `b = 5`. Write some statements to find the sum, difference, product and quotient of `a` and `b`.

Solutions to many of the exercises are in Appendix E.

2

MATLAB fundamentals

The objective of this chapter is to introduce some of the fundamentals of MATLAB programming, including:

- **variables, operators and expressions;**
- **arrays (including vectors and matrices);**
- **basic input and output;**
- **repetition (`for`);**
- **decisions (`if`).**

The tools introduced are sufficient to begin solving numerous scientific and engineering problems you may encounter in your course work and in your profession. Thus, the last part of this chapter and the next chapter describe an approach to designing reasonably good programs to initiate the building of tools in your own toolbox.

2.1 Variables and the workspace

2.1.1 Variables

Variables are fundamental to programming. In a sense, the art of programming is

getting the right values in the right variables at the right time.

A variable name (like the variable `balance` that we used in the previous chapter) must comply with the following two rules:

1. It may consist only of the letters `a–z`, the digits `0–9` and the underscore (`_`).
2. It must start with a letter.

A variable name may be as long as you like, but MATLAB only remembers the first 31 characters.

Examples of valid variable names: `r2d2` `pay_day`

Examples of invalid names (why?): `pay-day` `2a` `name$` `_2a`

A variable is created simply by assigning a value to it at the command line or in a program, e.g.

```
a = 98
```

If you attempt to refer to a non-existent variable you will get the error message

```
??? Undefined function or variable ...
```

The official MATLAB documentation refers to all variables as *arrays*, whether they are single-valued (scalars) or multi-valued (vectors or matrices). In other words, a scalar is a `1x1` array, i.e. an array with a single row and a single column which, of course, is an array of one item.

2.1.2 Case sensitivity

MATLAB is *case sensitive*, which means it distinguishes between upper- and lowercase letters. So `balance`, `BALANCE` and `BaLance` are three different variables.

Many programmers write variable names in lowercase except for the first letter of the second and subsequent words, if the name consists of more than one word run together. This style is known as *camel caps*, the uppercase letters looking like a camel's humps (with a bit of imagination). For example, `camelCaps`, `millenniumBug`, `dayOfTheWeek`. Other programmers prefer to separate words with underscores.

Command and function names are also case sensitive. However, note that when you use the command line `help`, function names are given in capitals, e.g. `CLC`, solely to emphasize them. You must *not* use capitals when running functions and commands!

2.1.3 The workspace

Another fundamental concept in MATLAB is the *workspace*. Enter the command `clear`, and then run the compound interest program again; see Section 1.3.2

in the previous chapter. Now enter the command `who`. You should see a list of variables as follows:

```
Your variables are:

balance     interest    rate
```

All the variables you create during a session remain in the workspace until you `clear` them. You can use or change their values at any stage during the session.

The command `who` lists the names of all the variables in your workspace.

The function `ans` returns the value of the last expression evaluated but not assigned to a variable.

The command `whos` lists the size of each variable as well:

```
Name            Size          Bytes  Class

balance         1x1               8  double array
interest        1x1               8  double array
rate            1x1               8  double array
```

Each variable here occupies eight *bytes* of storage. A byte is the amount of computer memory required for one character (if you are interested, one byte is the same as eight *bits*).

These variables each have a *size* of '1-by-1', because they are *scalars*, as opposed to vectors or matrices (although as mentioned above MATLAB regards them all as 1-by-1 arrays).

`double` means that the variable holds numeric values as double-precision floating point (see Section 2.4).

The command `clear` removes all variables from the workspace.

A particular variable can be removed from the workspace, e.g. `clear rate`.

More than one variable can also be cleared, e.g. `clear rate balance` (separate the variable names with spaces, *not commas!*).

When you run a program, any variables created by it remain in the workspace after it has run. This means that existing variables with the same names get overwritten.

The Workspace Browser in the desktop provides a handy visual representation of the workspace. You can view and even change the values of workspace variables with the Array Editor. To activate the Array Editor click on a variable in the Workspace Browser, or right-click to get the more general context menu. Note from the context menu that you can draw graphs of workspace variables in various ways.

2.1.4 Adding commonly used constants to the workspace

If you often use the same physical or mathematical constants in your MATLAB sessions, you can save them in an M-file and run it at the start of a session. For example, the following statements could be saved in `myconst.m`:

```
g = 9.8;                    % acceleration due to gravity
avo = 6.023e23;             % Avogadro's number
e = 2.718281828459045;      % base of natural log
pi_4 = pi / 4;
log10e = log10( e );
bar_to_kP = 101.325;        % atmospheres to kiloPascals
```

If you run `myconst` at the start of a session these six variables will be part of the workspace and will be available for the rest of the session, or until you `clear` them. This approach to using MATLAB is like a notepad (it is one of many ways). As your experience with this tool grows, you will discover many more utilities and capabilities associated with this computational and analytical environment.

2.2 Arrays: vectors and matrices

As mentioned earlier, the name MATLAB stands for MATrix LABoratory because MATLAB has been designed to work with *matrices*. A matrix is a rectangular object (e.g. a *table*) consisting of rows and columns. We will postpone most of the details of proper matrices and how MATLAB works with them until Chapter 6.

A *vector* is a special type of matrix, having only one row, or one column. Vectors are also called *lists* or *arrays* in other programming languages. If you haven't come across vectors officially yet, don't worry—just think of them as lists of numbers.

MATLAB handles vectors and matrices in the same way, but since vectors are easier to think about than matrices, we will look at them first. This will enhance

your understanding and appreciation of many other aspects of MATLAB. As mentioned above, MATLAB refers to scalars, vectors and matrices generally as arrays. We will also use the term array generally, with vector and matrix referring to their one-dimensional (1-D) and two-dimensional (2-D) forms.

2.2.1 Initializing vectors: explicit lists

To get going, try the following short exercises on the command line. (You won't need reminding about the command prompt ≫ any more, so it will no longer appear unless the context absolutely demands it.)

1. Enter a statement like

   ```
   x = [1 3 0 -1 5]
   ```

   Can you see that you have created a vector (list) with five *elements*? (Make sure to leave out the semicolon so that you can see the list. Also, make sure you hit Enter to execute the command.)

2. Enter the command `disp(x)` to see how MATLAB displays a vector.

3. Enter the command `whos` (or look in the Workspace browser). Under the heading `Size` you will see that `x` is 1-by-5, which means 1 row and 5 columns. You will also see that the total number of elements is 5.

4. You can put commas instead of spaces between vector elements if you like. Try this:

   ```
   a = [5,6,7]
   ```

5. Don't forget the commas (or spaces) between elements, otherwise you could end up with something quite different, e.g.

   ```
   x = [130-15]
   ```

   What do you think this gives?

6. You can use one vector in a list for another one, e.g. type in the following:

   ```
   a = [1 2 3];
   b = [4 5];
   c = [a -b];
   ```

   Can you work out what `c` will look like before displaying it?

7. And what about this?

```
a = [1 3 7];
a = [a 0 -1];
```

8. Enter the following

```
x = [ ]
```

and then note in the Workspace browser that the size of `x` is given as 0-by-0 because `x` is *empty*. This means `x` is defined, and can be used where an array is appropriate without causing an error; however, it has no size or value.

Making `x` empty is *not* the same as saying `x = 0` (in the latter case `x` has size 1-by-1), or `clear x` (which removes `x` from the workspace, making it undefined).

An empty array may be used to remove elements from an array (see Section 2.2.5 below).

These are all examples of the *explicit list* method of *initializing* vectors. Remember the following important rules:

- Elements in the list must be enclosed in *square* not round brackets.
- Elements in the list must be separated *either by spaces or by commas*.

2.2.2 Initializing vectors: the colon operator

A vector can also be generated (initialized) with the *colon operator*, as we have seen already in the previous chapter. Enter the following statements:

```
x = 1:10
```

(elements are the integers 1, 2, ..., 10);

```
x = 1:0.5:4
```

(elements are the values 1, 1.5, ..., 4 in increments of 0.5—note that if the colons separate three values, the *middle* value is the increment);

```
x = 10:-1:1
```

(elements are the integers 10, 9, ..., 1, since the increment is *negative*);

```
x = 1:2:6
```

(elements are 1, 3, 5; note that when the increment is positive but not equal to 1, the last element is not allowed to exceed the value after the second colon);

```
x = 0:-2:-5
```

(elements are $0, -2, -4$; note that when the increment is negative but not equal to -1, the last element is not allowed to be less than the value after the second colon);

```
x = 1:0
```

(a complicated way of generating an empty vector!).

2.2.3 `linspace`

The function `linspace` can be used to initialize a vector of equally spaced values, e.g.

```
linspace(0, pi/2, 10)
```

creates a vector of 10 equally spaced points from 0 to $\pi/2$ (inclusive).

2.2.4 Transposing vectors

All of the vectors examined so far are *row vectors*. Each has one row and several columns. To generate the *column vectors* that are often needed in mathematics, you need to *transpose* such vectors, i.e. you need to interchange their rows and columns. This is done with the single quote, or *apostrophe* (`'`), which is the nearest MATLAB can get to the mathematical dash (′) that is often used to indicate the transpose.

Enter `x = 1:5` and then enter `x'` to display the transpose of `x`. Note that `x` itself remains a row vector.

Or you can create a column vector directly:

```
y = [1 4 8 0 -1]'
```

2.2.5 Subscripts

We can refer to particular elements of a vector by means of *subscripts*. Try the following:

1. Enter `r = rand(1,7)`
   This gives you a row vector of seven *random numbers*.
2. Now enter `r(3)`
   This will display the *third* element of `r`. The number 3 is the *subscript*.
3. Now enter `r(2:4)`
   This should give you the second, third and fourth elements.
4. What about `r(1:2:7)`?
5. And `r([1 7 2 6])`?
6. You can use an empty vector to *remove* elements from a vector, e.g.

   ```
   r([1 7 2]) = [ ]
   ```

   will remove elements 1, 7 and 2.

To summarize:

- A subscript is indicated inside *round* brackets (also called 'parentheses').
- A subscript may be a scalar or a vector.
- In MATLAB subscripts always start at 1.
- Fractional subscripts are always rounded *down*, e.g. `x(1.9)` refers to element `x(1)`.

2.2.6 Matrices

A *matrix* may be thought of as a table consisting of rows and columns. You create a matrix just as you do a vector, except that a semicolon is used to indicate the end of a row, e.g. the statement

```
a = [1 2 3; 4 5 6]
```

results in

```
a =
     1     2     3
     4     5     6
```

A matrix may be transposed, e.g. with `a` initialized as above, the statement `a'` results in

```
a =
     1     4
     2     5
     3     6
```

A matrix can be constructed from column vectors of the same length. The statements

```
    x = 0:30:180;
table = [x' sin(x*pi/180)']
```

result in

```
table =
         0         0
   30.0000    0.5000
   60.0000    0.8660
   90.0000    1.0000
  120.0000    0.8660
  150.0000    0.5000
  180.0000    0.0000
```

2.2.7 Capturing output

You can use cut and paste techniques to tidy up the output from MATLAB statements, if this is necessary for some sort of presentation. Generate the table of angles and sines as shown above. Select all seven rows of numerical output in the Command Window. Copy the selected output to the Editor. You can then edit the output, e.g. by inserting text headings above each column (this is easier than trying to get headings to line up over the columns with a `disp` statement). The edited output can in turn be pasted into a report, or printed as it is (the **File** menu has a number of printing options).

Another way of capturing output is with the `diary` command. The command

`diary` *filename*

copies everything that subsequently appears in the Command Window to the text file *filename*. You can then edit the resulting file with any text editor (including the MATLAB Editor). Stop recording the session with

```
diary off
```

Note that `diary` *appends* material to an existing file, i.e. it adds new information to the end of the file.

2.3 Vertical motion under gravity

If a stone is thrown vertically upward with an initial speed u, its vertical displacement s after a time t has elapsed is given by the formula $s = ut - gt^2/2$, where g is the acceleration due to gravity. Air resistance has been ignored. We would like to compute the value of s over a period of about 12.3 seconds at intervals of 0.1 seconds, and to plot the distance–time graph over this period, as shown in Figure 2.1. The structure plan for this problem is as follows:

1. % Assign the data (g, u and t) to MATLAB variables.
2. % Calculate the value of s according to the formula.
3. % Plot the graph of s against t.
4. % Stop.

This plan may seem trivial to you, and a waste of time writing down. Yet you would be surprised how many beginners, preferring to rush straight to the computer, start with step 2 instead of step 1. It is well worth developing the mental discipline of structure-planning your program first. You could even use `cut` and `paste` to plan as follows:

1. Type the structure plan into the Editor (each line preceded by % as shown above).
2. Paste a second copy of the plan directly below the first.
3. Translate each line in the second copy into a MATLAB statement, or statements (add % comments as in the example below).
4. Finally, paste all the translated MATLAB statements into the Command Window, and run them.
5. If necessary, go back to the Editor to make corrections, and repaste the corrected statements to the Command Window (or save the program in the Editor as an M-file and execute it).

You might like to try this as an exercise, before looking at the final program, which is as follows:

```
% Vertical motion under gravity
g = 9.8;                        % acceleration due to
                                  gravity
```

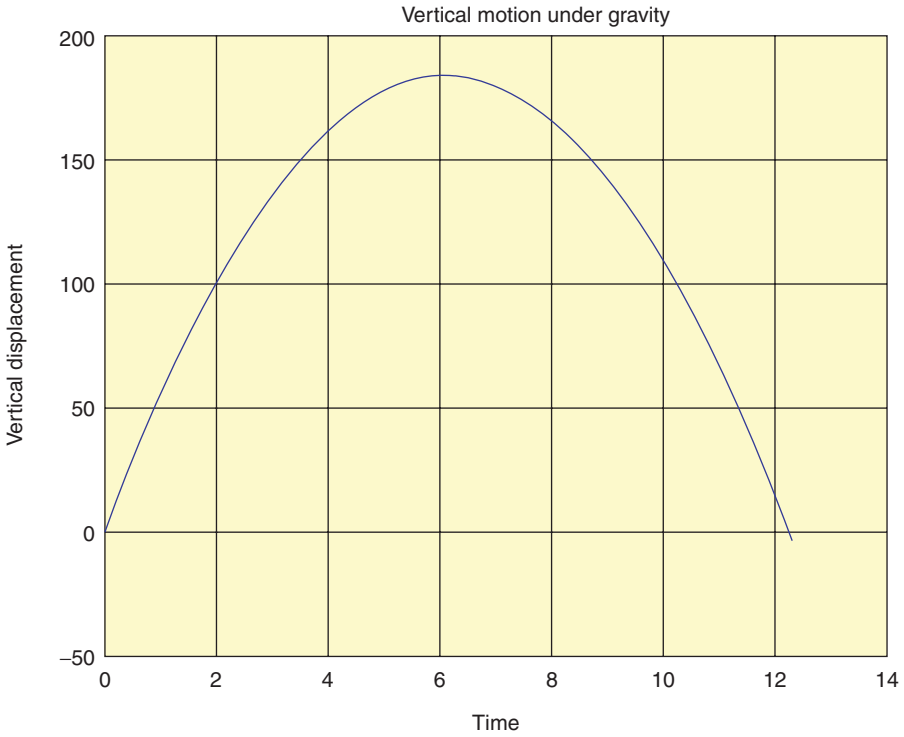


Figure 2.1 Distance–time graph of stone thrown vertically upward

```
u = 60;                          % initial velocity (meters/
                                   sec)
t = 0 : 0.1 : 12.3;              % time in seconds
s = u * t - g / 2 * t .^ 2;      % vertical displacement in
                                   meters

plot(t, s), title( 'Vertical motion under gravity' ), ...
     xlabel( 'time' ), ylabel( 'vertical displacement' ),
     grid
disp( [t' s'] )                  % display a table
```

The graphical output is shown in Figure 2.1.

Note the following points:

- Anything in a line following the percentage symbol `%` is ignored by MATLAB and may be used as a comment (description).
- The statement `t = 0 : 0.1 : 12.3` sets up a vector.
- The formula for `s` is evaluated *for every element* of the vector `t`, making another vector.

- The expression `t.^2` squares *each element* in `t`. This is called an *array operation*, and is different to squaring the vector itself, which is a *matrix* operation, as we shall see later.
- More than one statement can be entered on the same line if the statements are separated by commas.
- A statement or group of statements can be continued to the next line with an *ellipsis* of three or more dots `...`
- The statement `disp([t' s'])` first transposes the row vectors `t` and `s` into columns, and constructs a matrix from these two columns, which is then displayed.

You might want to save the program under a helpful name, like `throw.m` if you think you might come back to it. In that case, it would be worth keeping the structure plan as part of the file; just insert `%` symbols in front of each line of the structure plan. This way, the structure plan reminds you what the program does when you look at it again after some months. Note that you can use the context menu in the Editor to **Comment**/**Uncomment** a block of selected text.

2.4 Operators, expressions and statements

Any program worth its salt actually does something. What it basically does is to evaluate *expressions*, such as

```
u*t - g/2*t.^2
```

and to execute (carry out) statements, such as

```
balance = balance + interest
```

MATLAB has been described as "an *expression* based language. It interprets and evaluates typed expressions." Expressions are constructed from a variety of things, such as numbers, variables, and operators. First we need to look at numbers.

2.4.1 Numbers

Numbers can be represented in MATLAB in the usual decimal form (*fixed point*), with an optional decimal point, e.g.

```
1.2345    -123    .0001
```

A number may also be represented in *scientific notation*, e.g. 1.2345×10^9 may be represented in MATLAB as `1.2345e9`. This is also called *floating point* notation. The number has two parts: the *mantissa*, which may have an optional decimal point (`1.2345` in this example) and the *exponent* (`9`), which must be an integer (signed or unsigned). Mantissa and exponent must be separated by the letter `e` (or `E`). The mantissa is multiplied by the power of 10 indicated by the exponent.

Note that the following is *not* scientific notation: `1.2345*10^9`. It is actually an *expression* involving two arithmetic operations (`*` and `^`) and therefore more time consuming.

Use scientific notation if the numbers are very small or very large, since there's less chance of making a mistake, e.g. represent 0.000000001 as `1e-9`.

On computers using standard floating point arithmetic, numbers are represented to approximately 16 significant decimal digits. The *relative accuracy* of numbers is given by the function `eps`, which is defined as the distance between 1.0 and the next largest floating point number. Enter `eps` to see its value on your computer.

The range of numbers is roughly $\pm 10^{-308}$ to $\pm 10^{308}$. Precise values for your computer are returned by the MATLAB functions `realmin` and `realmax`.

EXERCISES

1. Enter the following numbers at the command prompt in scientific notation (answers are below):

$1.234 \times 10^5, \quad -8.765 \times 10^{-4}, \quad 10^{-15}, \quad -10^{12}.$

`(1.234e5, -8.765e-4, 1e-15, -1e12)`

2.4.2 Data types

MATLAB has 14 fundamental data types (or classes). The default numeric data type is double-precision; all MATLAB computations are done in double-precision. More information on data types can be found in the `Help` index.

MATLAB also supports signed and unsigned integer types and single-precision floating point, by means of functions such as `int8`, `uint8`, `single`, etc.

Table 2.1 Arithmetic operations between two scalars

Operation	Algebraic form	MATLAB
Addition	$a+b$	`a + b`
Subtraction	$a-b$	`a - b`
Multiplication	$a \times b$	`a * b`
Right division	a/b	`a / b`
Left division	b/a	`a \ b`
Power	a^b	`a ^ b`

However, before mathematical operations can be performed on such data types they must first be converted to double-precision using the `double` function.

2.4.3 Arithmetic operators

The evaluation of expressions is achieved by means of *arithmetic operators*. The arithmetic operations on two *scalar* constants or variables are shown in Table 2.1. Operators operate on *operands* (`a` and `b` in the table).

Left division seems a little curious: divide the right operand by the *left* operand. For *scalar* operands the expressions `1/3` and `3\1` have the same numerical value (a colleague of mine speaks the latter as '3 under 1'). However, *matrix* left division has an entirely different meaning, as we shall see later.

2.4.4 Precedence of operators

Several operations may be combined in one expression, e.g. `g * t ^ 2`. MATLAB has strict rules about which operations are performed first in such cases; these are called *precedence* rules. The precedence rules for the operators in Table 2.1 are shown in Table 2.2. Note that parentheses (round brackets) have the highest precedence. Note also the difference between round and square brackets. Round brackets are used to alter the precedence of operators, and to denote subscripts, while square brackets are used to create vectors.

When operators in an expression have the same precedence the operations are carried out from left to right. So `a / b * c` is evaluated as `(a / b) * c` and *not* as `a / (b * c)`.

Table 2.2 Precedence of arithmetic operations

Precedence	Operator
1	Parentheses (round brackets)
2	Power, left to right
3	Multiplication and division, left to right
4	Addition and subtraction, left to right

EXERCISES

1. Evaluate the following MATLAB expressions yourself before checking the answers in MATLAB:

```
1 + 2 * 3
4 / 2 * 2
1+2 / 4
1 + 2\4
2*2 ^ 3
2 * 3 \ 3
2 ^ (1 + 2)/3
1/2e-1
```

2. Use MATLAB to evaluate the following expressions. Answers are in brackets.

   (a) $\dfrac{1}{2 \times 3}$ (0.1667)

   (b) $2^{2 \times 3}$ (64)

   (c) $1.5 \times 10^{-4} + 2.5 \times 10^{-2}$ (0.0252; use scientific or floating point notation)

2.4.5 The colon operator

The colon operator has a lower precedence than + as the following shows:

```
1+1:5
```

The addition is carried out first, and then a vector with elements 2, ..., 5 is initialized.

Table 2.3 Arithmetic operators that operate element by element on arrays.

Operator	Description
`.*`	Multiplication
`./`	Right division
`.\`	Left division
`.^`	Power

You may be surprised at the following:

```
1+[1:5]
```

Were you? The value 1 is added to *each element* of the vector `1:5`. In this context, the addition is called an *array operation*, because it operates on each element of the vector (array). Array operations are discussed below.

See Appendix B for a complete list of MATLAB operators and their precedences.

2.4.6 The transpose operator

The transpose operator has the highest precedence. Try

```
1:5'
```

The 5 is transposed first (into itself since it is a scalar!), and then a row vector is formed. Use square brackets if you want to transpose the whole vector:

```
[1:5]'
```

2.4.7 Arithmetic operations on arrays

Enter the following statements at the command line:

```
a = [2 4 8];
b = [3 2 2];
a .* b
a ./ b
```

MATLAB has four further arithmetic operators, as shown in Table 2.3. These operators work on corresponding elements of arrays with equal dimensions.

The operations are sometimes called *array operations*, or *element-by-element* operations because the operations are performed element by element. For example, `a .* b` results in the following vector (sometimes called the *array product*):

```
[a(1)*b(1)  a(2)*b(2)  a(3)*b(3)]
```

i.e. `[6 8 10]`.

You will have seen that `a ./ b` gives element-by-element division.

Now try `[2 3 4] .^ [4 3 1]`. The *i*th element of the first vector is raised to the power of the *i*th element of the second vector.

The period (dot) is necessary for the array operations of multiplication, division and exponentiation, because these operations are defined differently for matrices; the operations are then called *matrix operations* (see Chapter 6).

With `a` and `b` as defined above, try `a + b` and `a - b`. For addition and subtraction, array operations and matrix operations are the same, so we don't need the period to distinguish between them.

When array operations are applied to two vectors, both vectors must be the same size!

Array operations also apply to operations between a scalar and a non-scalar. Check this with `3 .* a` and `a .^ 2`. This property is called *scalar expansion*. Multiplication and division operations between scalars and non-scalars can be written with or without the period, i.e. if `a` is a vector, `3 .* a` is the same as `3 * a`.

A common application of element-by-element multiplication is in finding the *scalar product* (also called the *dot* product) of two vectors **x** and **y**, which is defined as

$$\mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i.$$

The MATLAB function `sum(z)` finds the sum of the elements of the vector `z`, so the statement `sum(a .* b)` will find the scalar product of `a` and `b` (30 for `a` and `b` defined above).

EXERCISES

Use MATLAB array operations to do the following:

1. Add 1 to each element of the vector `[2 3 -1]`.
2. Multiply each element of the vector `[1 4 8]` by 3.
3. Find the array product of the two vectors `[1 2 3]` and `[0 -1 1]`. (Answer: `[0 -2 3]`)
4. Square each element of the vector `[2 3 1]`.

2.4.8 Expressions

An *expression* is a formula consisting of variables, numbers, operators, and function names. An expression is evaluated when you enter it at the MATLAB prompt, e.g. evaluate 2π as follows:

```
2 * pi
```

MATLAB's response is:

```
ans =
    6.2832
```

Note that MATLAB uses the function `ans` (which stands for **ans**wer) to return the last expression to be evaluated but not assigned to a variable.

If an expression is terminated with a semicolon (`;`) its value is not displayed, although it is still returned by `ans`.

2.4.9 Statements

MATLAB *statements* are frequently of the form

variable = *expression*

e.g.

```
s = u * t - g / 2 * t .^ 2;
```

This is an example of an *assignment* statement, because the value of the expression on the *right* is *assigned* to the variable (`s`) on the *left*. Assignment always works in this direction.

Note that the object on the left-hand side of the assignment must be a variable name. A common mistake is to get the statement the wrong way round, like

```
a + b = c
```

Basically any line that you enter in the Command Window or in a program, which MATLAB accepts, is a statement, so a statement could be an assignment, a command, or simply an expression, e.g.

```
x = 29;                 % assignment
clear                   % command
pi/2                    % expression
```

This naming convention is in keeping with most other programming languages and serves to emphasize the different types of statements that are found in programming. However, the MATLAB documentation tends to refer to all of these as 'functions'.

As we have seen, a semicolon at the end of an assignment or expression suppresses any output. This is useful for suppressing irritating output of intermediate results (or large matrices).

A statement which is too long to fit onto one line may be continued to the next line with an *ellipsis* of at least three dots, e.g.

```
x = 3 * 4 - 8 ....
    / 2 ^ 2;
```

Statements on the same line may be separated by commas (output not suppressed) or semicolons (output suppressed), e.g.

```
a = 2; b = 3, c = 4;
```

Note that the commas and semicolons are not technically part of the statements; they are *separators*.

Statements may involve array operations, in which case the variable on the left-hand side may become a vector or matrix.

2.4.10 Statements, commands and functions

The distinction between MATLAB *statements*, *commands* and *functions* can be a little fuzzy, since all can be entered on the command line. However, it is helpful to think of commands as changing the general environment in some way, e.g. `load`, `save`, and `clear`. Statements do the sort of thing we usually associate with programming, such as evaluating expressions and carrying out assignments, making decisions (`if`), and repeating (`for`). Functions return with calculated values or perform some operation on data, e.g. `sin`, `plot`.

2.4.11 Vectorization of formulae

With array operations you can easily evaluate a formula repeatedly for a large set of data. This is one of MATLAB's most useful and powerful features, and you should always be looking for ways of exploiting it.

Let us again consider, as an example, the calculation of compound interest. An amount of money A invested over a period of n years with an annual interest rate of r grows to an amount $A(1+r)^n$. Suppose we want to calculate final balances for investments of \$750, \$1000, \$3000, \$5000 and \$11 999, over 10 years, with an interest rate of 9 percent. The following program (`comp.m`) uses array operations on a vector of initial investments to do this.

```
format bank
A = [750 1000 3000 5000 11999];
r = 0.09;
n = 10;
B = A * (1 + r) ^ n;
disp( [A' B'] )
```

Output:

```
  750.00        1775.52
 1000.00        2367.36
 3000.00        7102.09
 5000.00       11836.82
11999.00       28406.00
```

Note:

1. In the statement `B = A * (1 + r)^ n`, the expression `(1 + r)^ n` is evaluated first, because exponentiation has a higher precedence than multiplication.

2. After that each element of the vector `A` is multiplied by the scalar `(1 + r) ^ n` (scalar expansion).
3. The operator `*` may be used instead of `.*` because the multiplication is between a scalar and a non-scalar (although `.*` would not cause an error because a scalar is a special case of an array).
4. A table is displayed, with columns given by the transposes of `A` and `B`.

This process is called *vectorization* of a formula. The operation in the statement described in item 1 above is such that every element in the vector `B` is determined by operating on every element of vector `A` all at once by interpreting once a single command line.

See if you can adjust the program `comp.m` to find the balances for a single amount `A` ($1000) over periods of 1, 5, 10, 15 and 20 years. **Hint:** Use a vector for `n`: `[1 5 10 15 20]`.

Danger!

I have seen quite a few beginners make the following mistake. Run the `comp.m` program, and then enter the statement

```
A(1+r)^n
```

at the prompt. The output is an unexpected warning followed by a *scalar*:

```
Warning: Subscript indices must be integer values.
ans =
  5.6314e+028
```

Any ideas? Well, the basic error is that the multiplication symbol `*` has been left out after the `A`. However, MATLAB still obligingly gives a result! This is because round brackets after the `A` signify an element of a vector, in this case element number `1+r`, which rounds down to 1, since `r` has the value 0.09 (fractional subscripts are always rounded down). `A(1)` has the value 750, so MATLAB obediently calculates 750^{10}.

The curious thing is that the same wrong answer is given even if `A` is a scalar, e.g. `A = 750`, because MATLAB treats a scalar as an array with one element (as `whos` demonstrates). *The moral is that you should develop the official MATLAB mindset of regarding all variables as 'arrays'.*

EXERCISES

1. Evaluate the following MATLAB expressions yourself (before you use MATLAB to check!). The numerical answers are in parentheses (or round brackets).

```
2 / 2 * 3                (3)
2 / 3 ^ 2                (2/9)
(2 / 3) ^ 2              (4/9)
2 + 3 * 4 - 4            (10)
2 ^ 2 * 3 / 4 + 3        (6)
2 ^ (2 * 3) / (4 + 3)    (64/7)
2 * 3 + 4                (10)
2 ^ 3 ^ 2                (64)
-4 ^ 2                   (−16; ^ has higher precedence than −)
```

2. Use MATLAB to evaluate the following expressions. The answers are in round brackets again.

a. $\sqrt{2}$ (1.4142; use `sqrt` or `^0.5`)

b. $\dfrac{3+4}{5+6}$ (0.6364; use brackets)

c. Find the sum of 5 and 3 divided by their product (0.5333)

d. 2^{3^2} (512)

e. Find the square of 2π (39.4784; use `pi`)

f. $2\pi^2$ (19.7392)

g. $1/\sqrt{2\pi}$ (0.3989)

h. $\dfrac{1}{2\sqrt{\pi}}$ (0.2821)

i. Find the cube root of the product of 2.3 and 4.5 (2.1793)

j. $\dfrac{1-\frac{2}{3+2}}{1+\frac{2}{3-2}}$ (0.2)

k. $1000(1+0.15/12)^{60}$ (2107.2, e.g. $1000 deposited for 5 years at 15 percent per year, with the interest compounded monthly)

l. $(0.0000123+5.678\times10^{-3})\times0.4567\times10^{-4}$ (2.5988×10^{-7}; use scientific notation, e.g. `1.23e-5` ...; do *not* use `^`)

3. Try to avoid using unnecessary brackets in an expression. Can you spot the errors in the following expression (test your corrected version with MATLAB):

```
(2(3+4)/(5*(6+1))^2
```

The MATLAB Editor has two useful ways of dealing with the problem of 'unbalanced delimiters' (which you should know about if you have been working through Help!):

- When you type a closing delimiter, i.e. a right `)`, `]` or `}`, its matching opening delimiter is briefly highlighted. So if you don't see the highlighting when you type a right delimiter you immediately know you've got one too many.
- When you position the cursor anywhere inside a pair of delimiters and select **Text -> Balance Delimiters** (or press **Ctrl+B**) the characters inside the pair of delimiters are highlighted.

4. Set up a vector `n` with elements 1, 2, 3, 4, 5. Use MATLAB array operations on the vector `n` to set up the following four vectors, each with five elements:

a. 2, 4, 6, 8, 10
b. 1/2, 1, 3/2, 2, 5/2
c. 1, 1/2, 1/3, 1/4, 1/5
d. $1, 1/2^2, 1/3^2, 1/4^2, 1/5^2$

5. Suppose `a` and `b` are defined as follows:

```
a = [2 -1 5 0];
b = [3 2 -1 4];
```

Evaluate by hand the vector `c` in the following statements. Check your answers with MATLAB.

a. `c = a - b;`
b. `c = b + a - 3;`
c. `c = 2 * a + a .^ b;`
d. `c = b ./ a;`
e. `c = b .\ a;`
f. `c = a .^ b;`
g. `c = 2.^ b+a;`
h. `c = 2*b/3.*a;`
i. `c = b*2.*a;`

6. Water freezes at 32° and boils at 212° on the Fahrenheit scale. If C and F are Celsius and Fahrenheit temperatures, the formula

$$F = 9C/5 + 32$$

converts from Celsius to Fahrenheit. Use the MATLAB command line to convert a temperature of 37°C (normal human temperature) to Fahrenheit (98.6°).

7. Engineers often have to convert from one unit of measurement to another; this can be tricky sometimes. You need to think through the process carefully. For example, convert 5 acres to hectares, given that an acre is 4840 square yards, a yard is 36 inches, an inch is 2.54 cm, and a hectare is 10000 m^2. The best

approach is to develop a formula to convert x acres to hectares. You can do this as follows.

one square yard = $(36 \times 2.54)^2$ cm^2
so one acre = $4840 \times (36 \times 2.54)^2$ cm^2
= 0.4047×10^8 cm^2
= 0.4047 hectares
so x acres = $0.4047x$ hectares

Once you have found the formula (but not until you have found the formula!), MATLAB can do the rest:

```
x = 5;              % acres
h = 0.4047 * x;     % hectares
disp( h )
```

8. Develop formulae for the following conversions, and use some MATLAB statements to find the answers, which are in brackets.

a. Convert 22 yards (an imperial cricket pitch) to meters. (20.117 meters)
b. One pound (weight) = 454 grams. Convert 75 kilograms to pounds. (165.20 pounds)
c. Convert 49 meters/second (terminal velocity for a falling person-shaped object) to km/hour. (176.4 km/hour)
d. One atmosphere pressure = 14.7 pounds per square inch (psi) = 101.325 kilo Pascals (kPa). Convert 40 psi to kPa. (275.71 kPa)
e. One calorie = 4.184 joules. Convert 6.25 kilojoules to calories. (1.494 kilocalories)

2.5 Output

There are two straightforward ways of getting output from MATLAB:

1. by entering a variable name, assignment or expression on the command line, without a semicolon;
2. with the `disp` statement, e.g. `disp( x )`.

2.5.1 `disp`

The general form of `disp` for a numeric variable is

`disp(` *variable* `)`

When you use `disp`, the variable name is not displayed, and you don't get a line feed before the value is displayed, as you do when you enter a variable name on the command line without a semicolon. `disp` generally gives a neater display.

You can also use `disp` to display a message enclosed in single quotes (called a *string*). Single quotes which are part of the message must be repeated, e.g.

```
disp( 'Pilate said, ''What is truth?''' );
```

To display a message and a numeric value on the same line use the following trick:

```
x = 2;
disp( ['The answer is ', num2str(x)] );
```

The output should be

```
The answer is 2
```

Square brackets create a vector, as we have seen. If we want to display a *string*, we create it, i.e. we type a message between apostrophes (or single quotes). This we have done already in the above example by defining the string `'The answer is '`. Note that the last space before the second apostrophe is part of the string. But *all* the components of a MATLAB array must either be numbers or strings (unless you use a *cell array*—see Chapter 11). So we convert the number `x` to its *string representation* with the function `num2str`; read this as 'number to string'.

You can display more than one number on a line as follows:

```
disp( [x y z] )
```

The square brackets create a vector with three elements, which are all displayed.

The command `more on` enables *paging* of output. This is very useful when displaying large matrices, e.g. `rand(100000,7)`. See `help more` for details. If you forget to switch on `more` before displaying a huge matrix you can always stop the display with **Ctrl+Break** or **Ctrl+C**.

2.5.2 `format`

The word *format* refers to how something is laid out: in this case MATLAB output. To begin with, MATLAB has the following two basic output rules:

1. It always attempts to display integers (whole numbers) exactly. However, if the integer is too large, it is displayed in scientific notation with five significant digits, e.g. 1234567890 is displayed as `1.2346e+009` (i.e. 1.2346×10^9). Check this by first entering 123456789 at the command line, and then 1234567890.

2. Numbers with decimal parts are displayed with four significant digits. If the value x is in the range $0.001 < x \leq 1000$ it is displayed in fixed point form, otherwise scientific (floating point) notation is used, in which case the *mantissa* is between 1 and 9.9999, e.g. 1000.1 is displayed as `1.0001e+003`. Check this by entering the following numbers at the prompt (on separate lines): 0.0011, 0.0009, 1/3, 5/3, 2999/3, 3001/3.

This is what is called the *default* format, i.e. what normally happens. However, you can change from the default with variations on the `format` command, as follows. If you want values displayed in scientific notation (*floating point* form) whatever their size, enter the command

```
format short e
```

All output from subsequent `disp` statements will be in scientific notation with five significant digits, until the next `format` command is issued. Enter this command and check it with the following values: 0.0123456, 1.23456, 123.456 (all on separate lines).

If you want more accurate output, you can use `format long e`. This also gives scientific notation, but with 15 significant digits. Try it out on 1/7.

Use `format long` to get fixed point notation with 15 significant digits, e.g. try 100/7 and `pi`.

If you're not sure of the order of magnitude of your output you can try `format short g` or `format long g`. The `g` stands for 'general'. MATLAB decides in each case whether to use fixed or floating point.

Use `format bank` for financial calculations; you get fixed point with two decimal digits (for the cents). Try it on 10000/7.

Suppress irritating line-feeds with `format compact`, which gives a more compact looking display. `format loose` reverts to the more airy display.

Use `format hex` to get hexadecimal display.

Use `format rat` to display a number as a rational approximation (ratio of two integers), e.g. `pi` is displayed as 355/113, a pleasant change from the tired old 22/7. Please note that even this is an *approximation*! Try out `format rat` on $\sqrt{2}$ and e (`exp(1)`).

The symbols `+`, `-` and a space are displayed for positive, negative and zero elements of a vector or matrix after the command `format +`. In certain applications this is a convenient way of displaying matrices.

The command `format` by itself reverts to the default format.

You can always try `help format` if you're confused!

You can also set the various `format` options by selecting **File -> Preferences** from the Command Window. The **Command Window Preferences** dialog box opens. Set your format preferences in the **Text display** box. Note that you can specify a host of preferences for all the MATLAB tools in this way (**File -> Preferences** from the tool window). MATLAB automatically saves your preferences for use in later sessions.

2.5.3 Scale factors

Enter the following commands (MATLAB's response is also shown).

```
≫ format compact
≫ x = [1e3 1 1e-4]
x =
  1.0e+003 *
    1.0000    0.0010    0.0000
```

With `format short` (the default) and `format long` a common *scale factor* is applied to the whole vector if its elements are very large, very small, or differ greatly in magnitude. In this example, the common scale factor is 1000, so the elements displayed must all be multiplied by this factor to get their proper value, e.g. for the second element `1.0e+003 * 0.0010` gives 1. Taking a factor of 1000 out of the third element (`1e-4`) leaves `1e-7`, which is represented by 0.0000 since only four decimal digits are displayed.

If you don't want a scale factor try `format bank` or `format short e` :

```
≫ format bank
≫ x
x =
       1000.00          1.00          0.00
≫ format short e
≫ x
x =
  1.0000e+003  1.0000e+000  1.0000e-004
```

2.6 Repeating with `for`

So far we have seen how to get data into a program (i.e. provide *input* to a program), how to do arithmetic, and how to get some results (i.e. get *output* from the program). In this section we look at a new feature: repetition. This is implemented by the extremely powerful `for` construct. We will first look at some examples of its use, followed by explanations.

For starters, enter the following group of statements on the command line. Enter the command `format compact` first to make the output neater:

```
for i = 1:5, disp(i), end
```

Now change it slightly to

```
for i = 1:3, disp(i), end
```

And what about

```
for i = 1:0, disp(i), end
```

Can you see what's happening? The `disp` statement is repeated five times, three times, and not at all.

2.6.1 Square roots with Newton's method

The square root x of any positive number a may be found using only the arithmetic operations of addition, subtraction and division with *Newton's method*. This is an iterative (repetitive) procedure that refines an initial guess.

To introduce in a rather elementary way the notion of *structured programming* (to be described in more detail in Chapter 3), let us consider the following plan. The structure plan of the algorithm to find the square root and a program with sample output for $a = 2$ are as follows.

The structure plan:

1. Initialize a
2. Initialize x to $a/2$
3. Repeat 6 times (say)
   Replace x by $(x + a/x)/2$
   Display x
4. Stop.

The program:

```
a = 2;
x = a/2;
disp(['The approach to sqrt(a) for a = ', num2str(a)])

for i = 1:6
  x = (x + a / x) / 2;
  disp( x )
end

disp( 'Matlab''s value: ' )
disp( sqrt(2) )
```

The output (after selecting `format long`) is as follows:

```
The approach to sqrt(a) for a = 2
    1.50000000000000
    1.41666666666667
    1.41421568627451
    1.41421356237469
    1.41421356237310
    1.41421356237310

Matlab's value:
   1.41421356237310
```

The value of x converges to a limit rather quickly in this case, which is $\sqrt{a}$. Note that it is identical to the value returned by MATLAB's `sqrt` function. Most

computers and calculators use a similar method internally to compute square roots and other standard mathematical functions.

The general form of Newton's method is presented in Chapter 17.

2.6.2 Factorials!

Run the following program to generate a list of n and $n!$ (spoken as 'n factorial', or 'n shriek') where

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n.$$

```
n = 10;
fact = 1;

for k = 1:n
    fact = k * fact;
    disp( [k fact] )
end
```

Do an experiment to find the largest value of n for which MATLAB can find n factorial. (You'd better leave out the `disp` statement! Or, you can move this statement from above the `end` command to below the `end` command.)

2.6.3 Limit of a sequence

The `for` loops are ideal for computing successive members of a sequence (as in Newton's method above). The following example also highlights a problem that sometimes occurs when computing a limit. Consider the sequence

$$x_n = a^n/n!, \quad n = 1, 2, 3, \ldots$$

where a is any constant, and $n!$ is the factorial function defined above. The question is: What is the limit of this sequence as n gets indefinitely large? Let's take the case $a = 10$. If we try to compute x_n directly we could get into trouble, because $n!$ gets large very rapidly as n increases, and numerical *overflow* could occur. However, the situation is neatly transformed if we spot that x_n is related to x_{n-1} as follows:

$$x_n = a x_{n-1}/n.$$

There are no numerical problems now. (This relationship between x_n and x_{n-1} is a typical kind of discovery by clever mathematicians. Engineers and scientists

certainly can be clever enough to discover these mathematical tricks; it is not, of course, expected knowledge but it is the kind of knowledge that you gain through experience and through literature searches designed to help you solve mathematical problems that arise in your work.)

The following program computes x_n for $a = 10$, and increasing values of n.

```
a = 10;
x = 1;
k = 20;            % number of terms

for n = 1:k
  x = a * x / n;
  disp( [n x] )
end
```

2.6.4 The basic `for` construct

In general the most common form of the `for` loop (for use in a program, not on the command line) is

`for` *index* = *j*:*k*
statements
`end`

or

`for` *index* = *j*:*m*:*k*
statements
`end`

Note the following points carefully:

1. $j:k$ is a vector with elements $j, j+1, j+2, \ldots, k$.
2. $j:m:k$ is a vector with elements $j, j+m, j+2m, \ldots$, such that the last element does not exceed k if $m > 0$, or is not less than k if $m < 0$.
3. *index* must be a variable. Each time through the loop it will contain the next element of the vector $j:k$ or $j:m:k$, and for each of these values *statements* (which may be one or more statement) are carried out.

   If the `for` construct has the form

   ```
   for k = first:increment:last
   ```

then the number of times the loop is executed may be calculated from the following equation:

$$\text{floor}\left(\frac{\text{last} - \text{first}}{\text{increment}}\right) + 1$$

where the MATLAB function `floor(x)` rounds `x` *down* towards $-\infty$. This value is called the *iteration count*, or *trip count*. As an example, let us consider the statement `for i = 1:2:6`. It has an iteration count of

$$\text{floor}\left(\frac{6-1}{2}\right) + 1 = \text{floor}\left(\frac{5}{2}\right) + 1 = 3.$$

Thus `i` takes the values 1, 3, 5. Note that if the iteration count is negative, the loop is not executed.

4. On completion of the `for` loop the index contains the last value used.

5. If the vector $j:k$ or $j:m:k$ is empty, *statements* are not executed, and control passes to the statement following the `end` statement.

6. The index does not have to appear explicitly in *statements*. It is basically a counter. In fact, if the index *does* appear explicitly in *statements*, the `for` can often be *vectorized* (more details on this are given in Section 2.6.7 below). A simple example of a more efficient (faster) program is as follows. The examples with `disp` at the beginning of this section were for the purposes of illustration only; strictly, it would be more efficient to say (without the word `for`)

```
i = 1:5; disp( i' )
```

Can you see the difference? In this case `i` is assigned as a vector (hence, this change *vectorizes* the original program).

7. It is good 'programming style' to *indent* (tabulate) the statements inside a `for` loop. You may have noticed that the Editor does this for you automatically. This feature is called *smart indenting*. Indenting preferences may be set from the Editor/Debugger's **File -> Preferences** menu. Expand the **Editor/Debugger** item in the **Preferences** menu and select the **Keyboard & Indenting Preferences** dialogue box. Note also that a block of highlighted text may be (un)indented with the Editor's **Text** menu. You will also have noticed that the Editor implements *syntax highlighting*. Keywords are in one color, strings in another, and so on. You can customize the syntax highlighting color scheme in the **Editor/Debugger Font & Colors Preferences** dialogue box. Click on **Set Colors** in the **Colors** box (with **Syntax highlighting** checked).

2.6.5 `for` in a single line

If you insist on using `for` in a single line, here is the general form:

`for` *index* = *j*`:`*k*`,` *statements*`,` `end`

or

`for` *index* = *j*`:`*m*`:`*k*`,` *statements*`,` `end`

Note:

1. Don't forget the commas (semicolons will also do if appropriate). If you leave them out you will get an error message.
2. Again, *statements* can be one or more statements, separated by commas or semicolons.
3. If you leave out `end`, MATLAB will wait for you to enter it. Nothing will happen until you do so.

2.6.6 More general `for`

A more general form of `for` is

`for` *index* = *v*

where *v* is any vector. The index moves through each element of the vector in turn. This provides a neat way of processing each item in a list. The most general form of the `for` loop will be discussed in Chapter 6; this is when we look at matrices in more detail.

2.6.7 Avoid `for` loops by vectorizing!

There are situations where a `for` loop is essential, as in many of the examples in this section so far. However, the way MATLAB has been designed, `for` loops tend to be inefficient in terms of computing time. If you have written a `for` loop which involves the index of the loop in an expression, it may be possible to vectorize the expression, making use of array operations where necessary, as the following examples show.

Suppose you want to evaluate

$$\sum_{n=1}^{100\,000} n$$

(and can't remember the formula for the sum!) Here's how to do it with a `for` loop (run the program, which also times how long it takes):

```
t0 = clock;
s = 0;
for n = 1:100000
    s = s + n;
end
etime(clock, t0)
```

The MATLAB function `clock` returns a six-element vector with the current date and time in the format year, month, day, hour, minute, seconds. So `t0` records when the calculation starts.

The function `etime` returns the time in seconds elapsed between its two arguments, which must be vectors as returned by `clock`. On the late Hahn's Pentium II, it returned about 3.35 seconds, which is the total time for this calculation. (If you have a faster PC, it should take less time.)

Now try to vectorize this calculation (before looking at the solution). Here it is:

```
t0 = clock;
n = 1:100000;
s = sum( n );
etime(clock, t0)
```

This way takes only 0.06 seconds on the same PC—more than 50 times faster!

There is a neater way of monitoring the time taken to interpret MATLAB statements: the `tic` and `toc` function (see below).

Now suppose you want to evaluate

$$\sum_{n=1}^{100\,000} \frac{1}{n^2}.$$

Here's the `for` loop version:

```
tic
s = 0;
for n = 1:100000
  s = s + 1/n^2;
end
toc
```

This takes about 6 seconds on the same PC. Once again, try to vectorize the sum:

```
tic
n = 1:100000;
s = sum( 1./n.^2 );
toc
```

The same PC gives a time of about 0.05 seconds for the vectorized version—more than 100 times faster! (Of course, the computation time in these examples is small regardless of the method applied. However, learning how to improve the efficiency of computation to solve more complex scientific or engineering problems will be helpful as you develop good programming skills—more details on good problem-solving and program-design practices are introduced at the end of this chapter and dealt with, in more detail, in the next chapter.)

Series with alternating signs are a little more challenging. This series sums to ln(2) (the natural logarithm of 2):

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \cdots$$

Here's how to find the sum of the first 9999 terms with a `for` loop (note how to handle the alternating sign):

```
sign = -1;
s = 0;

for n = 1:9999
  sign = -sign;
  s = s + sign / n;
end
```

Try it. You should get 0.6932. MATLAB's `log(2)` gives 0.6931. Not bad.

The vectorized version is as follows:

```
n = 1:2:9999;
s = sum( 1./n - 1./(n+1) )
```

If you time the two versions, you will again find that the vectorized form is many times faster.

MATLAB's functions naturally exploit vectorization wherever possible. For example, `prod(1:n)` will find $n!$ much faster than the code at the beginning of this section (for large values of n).

2.6.8 A common mistake: `for` less loops!

A very common mistake is to omit the word `for` from a `for` loop. Instead of reporting an error, MATLAB obligingly creates a vector, and executes the statements in the 'loop' only once. For example, run the program for square-rooting with Newton's method, leaving out the word `for`, i.e.

```
a = 2;
x = a/2;

i = 1:6                          % no for now
  x = (x + a / x) / 2;
  disp( x )
end
```

Can you explain the output? Check with `whos` that you have created a vector `i`. Note that the `end` statement by itself is perfectly legal, although it does precisely nothing.

EXERCISES

Write MATLAB programs to find the following sums (a) with `for` loops, and (b) by vectorization. Time both versions in each case.

1. $1^2 + 2^2 + 3^2 + \cdots + 1\,000^2$. (sum is 333 833 500)

2. $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots - \frac{1}{1003}$ (sum is 0.7849—converges slowly to $\pi/4$)

3. Sum the left-hand side of the series

$$\frac{1}{1^2 \cdot 3^2} + \frac{1}{3^2 \cdot 5^2} + \frac{1}{5^2 \cdot 7^2} + \cdots = \frac{\pi^2 - 8}{16}.$$ (sum is 0.1169, with 500 terms)

2.7 Decisions

The MATLAB function `rand` generates a random number in the range 0–1. Enter the following two statements at the command line:

```
r = rand
if r > 0.5 disp( 'greater indeed' ), end
```

MATLAB should only display the message `greater indeed` if `r` is in fact greater than 0.5 (check by displaying `r`). Repeat a few times—cut and paste from the Command History window (make sure that a new `r` is generated each time).

As a slightly different but related exercise, enter the following *logical expression* on the command line:

```
2 > 0
```

Now enter the logical expression `-1 > 0`. MATLAB gives a value of 1 to a logical expression which is *true*, and 0 to one which is *false*.

2.7.1 The one-line `if` statement

In the example above MATLAB has to make a decision; it must decide whether or not `r` is greater than 0.5. The `if` construct, which is fundamental to all computing languages, is the basis of such decision-making.

The simplest form of `if` in a single line is

`if` *condition statement*`, end`

Note the following points carefully:

1. *condition* is usually a *logical expression*, i.e. an expression containing a *relational operator*, and which is either *true* or *false*. The relational operators are shown in Table 2.4.

   MATLAB allows you to use an arithmetic expression for *condition*. If the expression evaluates to 0 it is regarded as false; any other value is true. This is not generally recommended; the `if` statement is easier (for you or a reader of your code) to understand if *condition* is a logical expression.

Table 2.4 Relational operators

Relational operator	Meaning
<	less than
<=	less than or equal
==	equal
~=	not equal
>	greater than
>=	greater than or equal

2. If *condition* is true, *statement* is executed, but if *condition* is false, nothing happens.
3. *condition* may be a vector or matrix, in which case it is true only if *all* its elements are non-zero. A single zero element in a vector or matrix renders it false.

Here are some more examples of logical expressions involving relational operators, with their meanings in brackets:

```
b^2 < 4*a*c        (b^2 < 4ac)
x >= 0             (x ≥ 0)
a ~= 0             (a ≠ 0)
b^2 == 4*a*c       (b^2 = 4ac)
```

Remember to use the double equal sign (==) when testing for equality, e.g.

```
if x == 0 disp( 'x equals zero' ), end
```

EXERCISES

The following statements all assign logical expressions to the variable x. See if you can correctly determine the value of x in each case before checking your answer with MATLAB.

```
x = 3 > 2
x = 2 > 3
```

```
x = -4 <= -3
x = 1 < 1
x = 2 ~= 2
x = 3 == 3
x = 0 < 0.5 < 1
```

Did you get the second-last one right? `3 == 3` is a logical expression which is true, since 3 is undoubtedly equal to 3. The value 1 (for true) is therefore assigned to `x`. And what about the last one? As a *mathematical* inequality,

$$0 < 0.5 < 1$$

is undoubtedly true fom a non-operational point of view. However, as a MATLAB operational expression, the left-hand `<` is evaluated first, i.e. `0 < 0.5`, giving `1` (true). Then the right-hand operation is performed, i.e. `1 < 1` giving `0` (false). Makes you think, doesn't it?

2.7.2 The `if-else` construct

Try the following exercises:

1. Enter the two lines

   ```
   x = 2;
   if x < 0 disp( 'neg' ), else disp( 'non-neg' ), end
   ```

   Did you get the message `non-neg`?

2. Now change the value of `x` to −1, and execute the `if` again. Did you get the message `neg` this time?

3. Now try

   ```
   if 79 disp( 'true' ), else disp( 'false' ), end
   ```

   Did you get `true`? Try some other values instead of 79, including 0 and some negative values.

4. Most banks offer differential interest rates. Suppose the rate is 9 percent if the amount in your savings account is less than $5000, but 12 percent otherwise. The Random Bank goes one step further, and gives you a random amount in your account to start with! Run the following program a few times:

   ```
   bal = 10000 * rand;
   ```

```
if bal < 5000
  rate = 0.09;
else
  rate = 0.12;
end

newbal = bal + rate * bal;
disp( 'New balance after interest compounded is:' )
format bank
disp( newbal )
```

Display the values of `bal` and `rate` each time from the command line to check that MATLAB has chosen the correct interest rate.

The basic form of `if-else` for use in a program file is

`if` *condition*
  statementsA
`else`
  statementsB
`end`

Note:

1. *statementsA* and *statementsB* represent one or more statements.
2. If *condition* is true, *statementsA* are executed, but if *condition* is false, *statementsB* are executed. This is essentially how you force MATLAB to choose between two alternatives.
3. The `else` part is optional.

2.7.3 The one-line `if-else` statement

The simplest general form of `if-else` for use on one line is

`if` *condition statementA*`,` *else statementB*`,` `end`

Note:

1. Commas (or semicolons) are essential between the various clauses.
2. The `else` part is optional.
3. Don't forget `end`, or MATLAB will wait forever.

2.7.4 elseif

Suppose the Random Bank now offers 9 percent interest on balances of less than $5000, 12 percent for balances of $5000 or more but less than $10 000, and 15 percent for balances of $10 000 or more. The following program calculates a customer's new balance after one year according to this scheme:

```
bal = 15000 * rand;

if bal < 5000
  rate = 0.09;
elseif bal < 10000
  rate = 0.12;
else
  rate = 0.15;
end

newbal = bal + rate * bal;
format bank
disp( 'New balance is:' )
disp( newbal )
```

Run the program a few times, and once again, display the values of `bal` and `rate` each time to convince yourself that MATLAB has chosen the correct interest rate.

In general, the `elseif` clause is used as follows:

```
if condition1
   statementsA
elseif condition2
   statementsB
elseif condition3
   statementsC
...
else
   statementsE
end
```

This is sometimes called an `elseif` *ladder*. It works as follows:

1. *condition1* is tested. If it is true, *statementsA* are executed; MATLAB then moves to the next statement after `end`.

2. If *condition1* is false, MATLAB checks *condition2*. If it is true, *statementsB* are executed, followed by the statement after `end`.
3. In this way, all the conditions are tested until a true condition is found. As soon as a true condition is found, no further `elseif`s are examined, and MATLAB jumps off the ladder.
4. If none of the conditions is true, *statementsE* after `else` are executed.
5. You should arrange the logic so that not more than one of the conditions is true.
6. There can be any number of `elseif`s, but at most one `else`.
7. `elseif` *must* be written as one word.
8. It is good programming style to indent each group of statements as shown.

2.7.5 Logical operators

More complicated logical expressions can be constructed using the three *logical operators* `&` (and), `|` (or), `~` (not). For example, the quadratic equation

$$ax^2 + bx + c = 0$$

has equal roots, given by $-b/(2a)$, provided that $b^2 - 4ac = 0$ *and* $a \neq 0$ (Figure 2.2). This translates to the following MATLAB statements:

```
if (b^2 - 4*a*c == 0) & (a ~= 0)
  x = -b / (2*a);
end
```

Of course, `a`, `b`, and `c` must be assigned values prior to reaching this set of statements. Note the double equal sign in the test for equality; see Chapter 5 for more on logical operators.

2.7.6 Multiple `if`s versus `elseif`

You could have written the Random Bank program as follows:

```
bal = 15000 * rand;

if bal < 5000
    rate = 0.09;
```

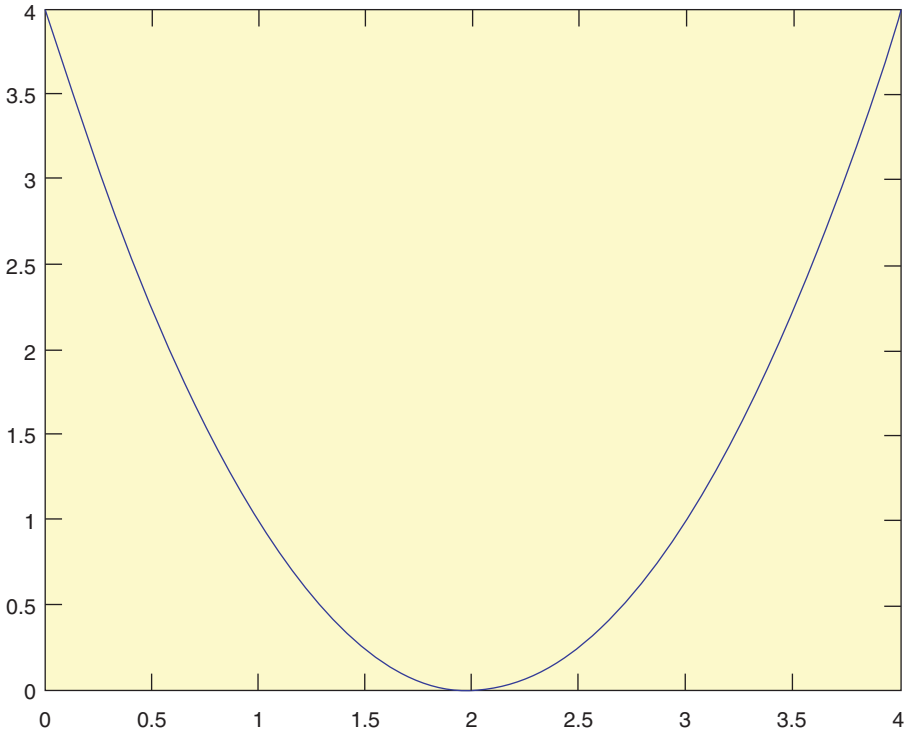


Figure 2.2 A quadratic function with equal roots

```
end
if bal >= 5000 & bal < 10000
    rate = 0.12;
end
if bal >= 10000
    rate = 0.15;
end

newbal = bal + rate * bal;
format bank
disp( 'New balance is' )
disp( newbal )
```

However, this is inefficient since each of the three conditions is always tested, even if the first one is true. In the earlier `elseif` version, MATLAB jumps off the `elseif` ladder as soon as it finds a true condition. This could save a lot of computing time (and is easier to read) if the `if` construct is in a loop which is repeated often.

Using this form, instead of the `elseif` ladder, you can make the following common mistake:

```
if bal < 5000
    rate = 0.09;
end
if bal < 10000
    rate = 0.12;
end
if bal >= 10000
    rate = 0.15;
end
```

Can you see why you get the wrong answer (1120 instead of 1090) if `bal` has the value 1000? When designing the logic you need to make sure that one and only one of the conditions will be true at any one time.

Another mistake I have seen students frequently make is to replace the second `if` with something like

```
if 5000 < bal < 10000
    rate = 0.12;
end
```

which is compelling as we saw above. However, *whatever* the value of `bal` is this condition will *always* be true. Can you see why?

2.7.7 Nested `if`s

An `if` construct can contain further `if`s, and so on. This is called *nesting*, and should not be confused with the `elseif` ladder. You have to be rather careful with the `else`s. In general, `else` belongs to the most recent `if` which has not been `end`ed. The correct positioning of `end` is therefore very important, as the next example demonstrates.

Suppose you want to compute the solution of a quadratic equation. You may want to check whether $a = 0$, to prevent a division by zero. Your program could contain the following nested `if`s:

```
...
d = b^2 - 4*a*c;
if a ~= 0
  if d < 0
```

```
    disp( 'Complex roots' )
  else
    x1 = (-b + sqrt( d )) / (2*a);
    x2 = (-b - sqrt( d )) / (2*a);
  end    % first end  <<<<<<<<<<<<
end
```

The `else` belongs to the second `if` by default, as intended.

Now move the first `end` up as follows:

```
d = b^2 - 4*a*c;
if a ~= 0
  if d < 0
    disp( 'Complex roots' )
  end    % first end moved up now  <<<<<<<<<<<<
  else
    x1 = (-b + sqrt( d )) / (2*a);
    x2 = (-b - sqrt( d )) / (2*a);

end
```

The `end` which has been moved now closes the second `if`. The result is that `else` belongs to the *first* `if` instead of to the second one. Division by zero is therefore guaranteed, instead of prevented!

2.7.8 Vectorizing `if`s?

You may be wondering if `for` statements enclosing `if`s can be vectorized. The answer is yes, by courtesy of *logical arrays*. Discussion of this rather interesting topic is postponed to Chapter 5.

2.7.9 `switch`

The `switch` statement executes certain statements based on the value of a variable or expression. In this example it is used to decide whether a random integer is 1, 2 or 3 (see **Rolling dice** in Chapter 5 for an explanation of this use of `rand`):

```
d = floor(3*rand) + 1
switch d
case 1
   disp( 'That''s a 1!' );
```

```
case 2
   disp( 'That''s a 2!' );
otherwise
   disp( 'Must be 3!' );
end
```

Multiple expressions can be handled in a single `case` statement by enclosing the case expression in a cell array (see Chapter 11):

```
d = floor(10*rand);
switch d
case {2, 4, 6, 8}
   disp( 'Even' );
case {1, 3, 5, 7, 9}
   disp( 'Odd' );
otherwise
    disp( 'Zero' );
end
```

2.8 Complex numbers

If you are not familiar with *complex* numbers, you can safely omit this section. However, it is useful to know what they are since the square root of a negative number may come up as a mistake if you are trying to work only with *real* numbers.

It's very easy to handle complex numbers in MATLAB. The special values `i` and `j` stand for $\sqrt{-1}$. Try `sqrt(-1)` to see how MATLAB represents complex numbers.

The symbol `i` may be used to assign complex values, e.g.

```
z = 2 + 3*i
```

represents the complex number $2+3i$ (*real part* 2, *imaginary part* 3). You can also input a complex value like this, e.g. enter

```
2 + 3*i
```

in response to the `input` prompt (remember, no semicolon). The imaginary part of a complex number may also be entered without an asterisk, e.g. `3i`.

All the arithmetic operators (and most functions) work with complex numbers, e.g. `sqrt(2 + 3*i)`, `exp(i*pi)`.

There are some functions that are specific to complex numbers. If `z` is a complex number `real(z)`, `imag(z)`, `conj(z)` and `abs(z)` all have the obvious meanings.

A complex number may be represented in polar coordinates, i.e.

$$z = re^{i\theta}.$$

`angle(z)` returns θ between $-\pi$ and π, i.e. it returns `atan2(imag(z), real(z))`. `abs(z)` returns the magnitude r.

Since $e^{i\theta}$ gives the unit circle in polars, complex numbers provide a neat way of plotting a circle. Try the following:

```
circle = exp( 2*i*[1:360]*pi/360 );
plot(circle)
axis('equal')
```

Note:

- If `y` is complex, the statement `plot(y)` is equivalent to

  ```
  plot(real(y), imag(y))
  ```

- The statement `axis('equal')` is necessary to make circles look round; it changes what is known as the *aspect ratio* of the monitor. `axis('normal')` gives the default aspect ratio.

If you are using complex numbers, be careful not to use `i` or `j` for other variables; the new values will replace the value of $\sqrt{-1}$, and will cause nasty problems!

For complex matrices, the operations `'` and `.'` behave differently. The `'` operator is the *complex conjugate transpose*, meaning rows and columns are interchanged, and signs of imaginary parts are changed. The `.'` operator, on the other hand, does a pure transpose without taking the complex conjugate. To see this, set up a complex matrix `a` with the statement

```
a = [1+i 2+2i; 3+3i 4+4i]
```

which results in

```
a =
   1.0000 + 1.0000i   2.0000 + 2.0000i
   3.0000 + 3.0000i   4.0000 + 4.0000i
```

The statement

```
a'
```

then results in the complex conjugate transpose

```
ans =
   1.0000 - 1.0000i   3.0000 - 3.0000i
   2.0000 - 2.0000i   4.0000 - 4.0000i
```

whereas the statement

```
a.'
```

results in the pure transpose

```
ans =
   1.0000 + 1.0000i   3.0000 + 3.0000i
   2.0000 + 2.0000i   4.0000 + 4.0000i
```

2.9 More on input and output

This section is not strictly 'essential' MATLAB; you can skip it and come back to it at a later time in your investigation of MATLAB.

2.9.1 `fprintf`

If you are the sort of person who likes to control *exactly* what your output looks like, then this section is for you. Otherwise you can happily stay with `disp`, `format` and cut-and-paste.

The `fprintf` statement is much more flexible (and therefore more complicated!) than `disp`. For example, it allows you to mix strings and numbers freely on the same line, and to control the format (e.g. number of decimal places) completely. As an example, change your compound interest program of Section 1.3.2 to look as follows:

```
balance = 12345;
rate = 0.09;
interest = rate * balance;
```

```
balance = balance + interest;
fprintf( 'Interest rate: %6.3f   New balance:
          %8.2f\n', ... rate, balance );
```

When you run this, your output should look like this:

```
Interest rate:  0.090   New balance: 13456.05
```

The most common form of `fprintf` is

`fprintf( '`*format string*`' ,` *list of variables* `)`

Note:

1. *format string* may contain a message. It may also contain *format specifiers* of the form `%e`, `%f` or `%g`, which control how the variables listed are embedded in the format string.
2. In the case of the `e` and `f` specifiers, the field width and number of decimal places or significant digits may be specified immediately after the `%`, as the next two examples show.
   - `%8.3f` means fixed point over 8 columns altogether (including the decimal point and a possible minus sign), with 3 decimal places (spaces are filled in from the left if necessary). Use 0 if you don't want any decimal places, e.g. `%6.0f`. Use leading zeros if you want leading zeros in the output, e.g. `%03.0f`.
   - `%12.2e` means scientific notation over 12 columns altogether (including the decimal point, a possible minus sign, and five for the exponent), with 2 digits in the mantissa after the decimal point (the mantissa is always adjusted to be not less than 1).
3. The `g` specifier is mixed, and leaves it up to MATLAB to decide exactly what format to use. This is a good one to use if you can't be bothered to count decimal places carefully, and/or aren't sure of the approximate magnitude of your result.
4. In the case of a vector, the sequence of format specifiers is repeated until all the elements have been displayed.
5. The *format string* in `fprintf` may also contain *escape codes* such as `\n` (line feed), `\t` (horizontal tab), `\b` (backspace) and `\f` (form feed).

A C programmer will no doubt feel very much at home here! For more details see `fprintf` in the online Help.

2.9.2 Output to a disk file with `fprintf`

Output may be sent to a disk file with `fprintf`. The output is *appended*, i.e. added on at the end of the file.

The general form is

`fprintf( 'filename', 'format string', list of variables )`

e.g.

```
fprintf( 'myfile', '%g', x )
```

sends the value of `x` to the disk file `myfile`.

2.9.3 General file I/O

MATLAB has a useful set of file I/O functions, such as `fopen`, `fread`, `fwrite`, `fseek`, etc., which are discussed in Chapter 4.

2.9.4 Saving and loading data

All or part of the workspace can be saved to a disk file and loaded back in a later session with the `save` and `load` commands. See Chapter 4 for details on these and other ways of importing and exporting MATLAB data.

2.10 Odds 'n ends

2.10.1 Variables, functions and scripts with the same name

Enter the command `rand`. You will get a random number; in fact, a different one each time you enter the command. Now enter the statement

```
rand = 13;
```

The name `rand` now represents a variable with the value 13 (check with `whos`). The random number generator `rand` has been hidden by the variable of the same name, and is unavailable until you `clear rand`. A script file can also get hidden like this.

When you type a name at the command-line prompt, say `goo`, the MATLAB interpreter goes through the following steps:

1. Looks for `goo` as a variable.
2. Looks in the current directory for a script file called `goo.m`.
3. Looks for `goo` as a built-in function, like `sin` or `pi`.
4. Looks (in order) in the directories specified by MATLAB's search path for a script file called `goo.m`. Use **File -> Set Path** to view and change MATLAB's search path.

I have seen students accidentally hiding scripts in this way during hands-on tests—it is a traumatic experience! If you are worried that there might be a MATLAB function `goo` lurking around to hide your script of the same name, first try `help goo`. If you get the message `goo.m not found`, then you're safe. To unhide a hidden `goo` type `clear goo`.

2.10.2 The `input` statement

Rewrite the script file `compint.m` carefully so that it looks exactly like this, and remember to save it:

```
balance = input( 'Enter bank balance: ' );
rate = input( 'Enter interest rate: ' );
interest = rate * balance;
balance = balance + interest;
format bank
disp( 'New balance:' );
disp( balance );
```

Enter the script file name at the prompt, and enter values of 1000 and 0.15 for the balance and interest rate respectively when asked to, so that your Command Window contains the following lines:

```
≫ compint

Enter bank balance: 1000

Enter interest rate: 0.15
New balance:
       1150.00
```

The `input` statement provides a more flexible way of getting data into a program than by assignment statements which need to be edited each time the data must be changed. It allows you to enter data *while a script is running*. The general form of the `input` statement is:

variable = `input( 'prompt' );`

Note:

- *prompt* is a message which prompts the user for the value(s) to be entered. It must be enclosed in apostrophes (single quotes).
- A semicolon at the end of the `input` statement will prevent the value entered from being immediately echoed on the screen.
- You wouldn't normally use `input` from the command line, since you really shouldn't need to prompt yourself in command-line mode!
- Vectors and matrices may also be entered with `input`, as long as you remember to enclose the elements in square brackets.
- You can enter an *expression* in response to the prompt, e.g. `a + b` (as long as `a` and `b` have been defined) or `rand(5)`. When entering an expression in this way don't include a semicolon (it is not part of the expression).

EXERCISES

1. Rewrite the solutions to a few of the unit conversion problems in Section 2.4 using `input` statements in script files.
2. Rewrite the program `comp.m` in **Vectorization of formulae** at the end of Section 2.4 to `input` the initial investment vector.

2.10.3 Shelling out to the operating system

You can *shell out* of MATLAB to the operating system by prefacing an operating system command with an exclamation mark (bang). For example, suppose you suddenly need to reset the system time during a MATLAB session. You can do this from the Command Window with the command

```
!time
```

(under Windows). Shelling out is achieved without quitting the current MATLAB session.

2.10.4 More Help functions

By now you should be finding your way around the online Help. Here are some alternative command-line ways of getting help. The command `doc` *function* displays the reference page for *function* in the Help browser, providing syntax, a description, examples and links to related functions. The command `helpwin` displays in the Help browser a list of all functions, with links to M-file help for each function.

2.11 Programming style

Some programmers delight in writing terse and obscure code; there is at least one annual competition for the most incomprehensible C program. A large body of responsible programmers, however, believe it is extremely important to develop the art of writing programs which are well laid out, with all the logic clearly described. Serious programmers therefore pay a fair amount of attention to what is called *programming style*, in order to make their programs clearer and more readable both to themselves, and to other potential users. You may find this irritating, if you are starting to program for the first time, because you will naturally be impatient to get on with the job. But a little extra attention to your program layout will pay enormous dividends in the long run, especially when it comes to debugging.

Some hints on how to improve your programming style are given below.

- You should make liberal use of comments, both at the beginning of a script, to describe briefly what it does and any special methods that may have been used, and also throughout the coding to introduce different logical sections.
- The meaning of each variable should be described briefly in a comment when it is initialized. You should describe variables systematically, e.g. in alphabetical order.
- Blank lines should be freely used to separate sections of coding (e.g. before and after loop structures).
- Coding (i.e. statements) inside structures (`for`s, `if`s, `while`s) should be indented (tabulated) a few columns to make them stand out.
- Blank spaces should be used in expressions to make them more readable, e.g. on either side of operators and equal signs. However, blanks may be omitted in places in complicated expressions, where this may make the logic clearer.

Computer programs are like any other literary art form; they should be designed and written with a reasonable amount of forethought leading to a clear statement of the problem that is to be solved to satisfy a recognized need for a scientific, technical, engineering or mathematical solution. The solution could be intended to enhance knowledge or to help the scientist or engineer make a decision about a proposed design of laboratory equipment or an artifact to enhance our standard of living.

The development of the problem statement is, probably, the most difficult part of any design process; it is no different for the design of a computer program. Hence, learning to design good computer programs (or codes, as they are sometimes called) provides good experience in the practice of creative design. A strategic plan is required that leads to the development of an *algorithm* (i.e. the sequence of operations required for solving a problem in a finite number of steps) to be programmed for MATLAB to execute in order to provide an answer to the problem posed. The essential goal is to create a *top-down* structure plan itemizing all of the steps of the algorithm that must be implemented in MATLAB to obtain the desired solution. The methodology of developing such a plan is described in the next chapter.

Summary

- The MATLAB desktop consists of a number of tools: the Command Window, the Workspace browser, the Current Directory browser and the Command History window.
- MATLAB has a comprehensive online Help system. It can be accessed through the help button in the desktop toolbar, or the **Help** menu in any tool.
- A MATLAB program can be written in the Editor and cut and pasted to the Command Window.
- A script file is a text file (created by the MATLAB Editor, or any other text editor) containing a collection of MATLAB statements, i.e. a program.

  The statements are carried out when the script file name is entered at the prompt in the Command Window.

  A script file name must have the `.m` extension. Script files are therefore also called M-files.
- The recommended way to run a script is from the Current Directory browser. The output from the script will then appear in the Command Window.

- A variable name consists only of letters, digits and underscores, and must start with a letter. Only the first 31 characters are significant. MATLAB is case-sensistive by default.

  All variables created during a session remain in the workspace, until removed with `clear`.

  The command `who` lists the variables in the workspace. `whos` also gives their sizes.

- MATLAB refers to all variables as *arrays*, whether they are scalars (single-valued arrays), vectors (1-D arrays) or matrices (2-D arrays).

- MATLAB names are case-sensitive.

- The Workspace browser in the desktop provides a handy visual representation of the workspace. Clicking a variable in the Workspace browser invokes the Array Editor which may be used to view and change variable values.

- Vectors and matrices are entered in square brackets. Elements are separated by spaces or commas. Rows are separated by semicolons.

  The colon operator is used to generate vectors with elements increasing (decreasing) by regular increments (decrements).

  Vectors are row vectors by default. Use the apostrophe transpose operator (`'`) to change a row vector into a column vector.

- An element of a vector is referred to by a subscript in round brackets.

  A subscript may itself be a vector.

  Subscripts always start at 1, and are rounded down if non-integer.

- The `diary` command copies everything that subsequently appears in the Command Window to the specified text file until the command `diary off` is given.

- Statements on the same line may be separated by commas or semicolons.

- A statement may be continued to the next line with an ellipsis of at least three dots.

- Numbers may be represented in fixed point decimal notation or in floating point scientific notation.

- MATLAB has 14 data types. The default numeric type is double-precision. All mathematical operations are carried out in double-precision.

- The six arithmetic operators for scalars are `+ - * \ / ^` . They operate according to rules of precedence. Brackets have the highest precedence.

- An expression is a rule for evaluating a formula using numbers, operators, variables and functions.

  A semicolon after an expression suppresses display of its value.

- Array operations are element-by-element operations between vectors, or between scalars and vectors.

  The array operations of multiplication, right and left division, and exponentiation are indicated by `.*` `./` `.\` `.^` to distinguish them from the matrix operations of the same name.

  Array operations may be used to evaluate a formula repeatedly for some or all of the elements of a vector. This is called vectorization of the formula.

- `disp` is used to output (display) numbers and strings.

  `num2str` is useful with `disp` for displaying strings and numbers on the same line.

  The `format` command controls the way output is displayed. Format may also be set by **File -> Preferences -> Command Window Preferences**.

- When vectors are displayed, a common scale factor is used if the elements are very large, very small, or differ greatly in magnitude.

- The `for` statement is used to repeat a group of statements a fixed number of times.

  If the index of a `for` statement is used in the expression being repeated, the expression can often be vectorized, saving a great deal of computing time.

- `tic` and `toc` may be used as a stopwatch.

- Logical expressions have the value true (1) or false (0) and are constructed with the six relational operators `>` `>=` `<` `<=` `==` `~=` .

  Any expression which evaluates to zero is regarded as false. Any other value is true.

  More complicated logical expressions can be formed from other logical expressions using the logical operators `&` (and), `|` (or), `~` (not).

- `if-else` executes different groups of statements according to whether a logical expression is true or false.

  The `elseif` ladder is a good way to choose between a number of options, only one of which should be true at a time.

- `switch` enables choices to be made between discrete cases of a variable or expression.

- A string is a collection of characters enclosed in single quotes (apostrophes).

- Complex numbers may be represented using the special variables `i` and `j`, which stand for the unit imaginary number $\sqrt{-1}$.

- `fprintf` is used to control output format more precisely.

- `save` and `load` are used to export and import data.

- The `input` statement, with a prompt, is used to prompt the user for input from the keyboard while a script is executing.

- MATLAB first checks whether a name is a variable, then a built-in function, then a script. Use `clear` to unhide a function or script if necessary.
- Operating system commands can be executed from the MATLAB prompt by typing an exclamation mark in front of them, e.g. `!copy`. This is called shelling out.
- Attention should be paid to programming style when writing scripts.

EXERCISES

2.1 Decide which of the following numbers are not acceptable in MATLAB, and state why not:

(a) `9,87` (b) `.0` (c) `25.82` (d) `-356231`
(e) `3.57*e2` (f) `3.57e2.1` (g) `3.57e+2` (h) `3,57e-2`

2.2 State, giving reasons, which of the following are not valid MATLAB variable names:

(a) `a2` (b) `a.2` (c) `2a` (d) `'a'one`
(e) `aone` (f) `_x_1` (g) `miXedUp` (h) `pay day`
(i) `inf` (j) `Pay_Day` (k) `min*2` (l) `what`

2.3 Translate the following expressions into MATLAB:

$p + \frac{w}{u}$

$p + \frac{w}{u+v}$

$\frac{p + \frac{w}{u+v}}{p + \frac{w}{u-v}}$

$x^{1/2}$

y^{y+z}

x^{y^z}

$(x^y)^z$

$x - x^3/3! + x^5/5!$

2.4 Translate the following into MATLAB statements:
Add 1 to the value of `i` and store the result in `i`.
Cube `i`, add `j` to this, and store the result in `i`.
Set `g` equal to the larger of the two variables `e` and `f`.
If `d` is greater than zero, set `x` equal to minus `b`.
Divide the sum of `a` and `b` by the product of `c` and `d`, and store the result in `x`.

2.5 What's wrong with the following MATLAB statements?

```
n + 1 = n;
```

```
Fahrenheit temp = 9*C/5 + 32;
2 = x;
```

2.6 Write a program to calculate x, where

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

and $a = 2$, $b = -10$, $c = 12$ (Answer 3.0)

2.7 There are eight pints in a gallon, and 1.76 pints in a liter. The volume of a tank is given as 2 gallons and 4 pints. Write a script which inputs this volume in gallons and pints and converts it to liters. (Answer: 11.36 liters)

2.8 Write a program to calculate petrol consumption. It should assign the distance traveled (in kilometers) and the amount of petrol used (in liters) and compute the consumption in km/liter as well as in the more usual form of liters per 100 km. Write some helpful headings, so that your output looks something like this:

```
Distance     Liters used     km/L      L/100km
528          46.23           11.42     8.76
```

2.9 Write some statements in MATLAB which will exchange the contents of two variables `a` and `b`, using only one additional variable `t`.

2.10 Try Exercise 2.9 *without* using any additional variables!

2.11 If C and F are Celsius and Fahrenheit temperatures respectively, the formula for conversion from Celsius to Fahrenheit is $F = 9C/5 + 32$.

(a) Write a script which will ask you for the Celsius temperature and display the equivalent Fahrenheit one with some sort of comment, e.g.

```
The Fahrenheit temperature is: ...
```

Try it out on the following Celsius temperatures (answers in brackets): 0 (32), 100 (212), −40 (−40!), 37 (normal human temperature: 98.6).

(b) Change the script to use vectors and array operations to compute and display the Fahrenheit equivalent of Celsius temperatures ranging from 20° to 30° in steps of 1°, in two columns with a heading, e.g.

```
Celsius      Fahrenheit
20.00        68.00
21.00        69.80
...
30.00        86.00
```

2.12 Generate a table of conversions from degrees (first column) to radians (second column). Degrees should go from 0° to 360° in steps of 10°. Recall that π radians = 180°.

2.13 Set up a matrix (table) with degrees in the first column from 0 to 360 in steps of 30, sines in the second column, and cosines in the third column. Now try to add tangents in the fourth column. Can you figure out what's going on? Try some variations of the `format` command.

2.14 Write some statements that display a list of integers from 10 to 20 inclusive, each with its square root next to it.

2.15 Write a single statement to find and display the sum of the successive *even* integers 2, 4, . . . , 200. (Answer: 10 100)

2.16 Ten students in a class write a test. The marks are out of 10. All the marks are entered in a MATLAB vector `marks`. Write a statement to find and display the average mark. Try it on the following marks:

```
5 8 0 10 3 8 5 7 9 4
```
(Answer: 5.9)

Hint: Use the function `mean`.

2.17 What are the values of `x` and `a` after the following statements have been executed?

```
a = 0;
i = 1;
x = 0;
a = a + i;
x = x + i / a;
a = a + i;
x = x + i / a;
a = a + i;
x = x + i / a;
a = a + i;
x = x + i / a;
```

2.18 Rewrite the statements in Exercise 2.17 more economically by using a `for` loop. Can you do even better by vectorizing the code?

2.19 (a) Work out by hand the output of the following script for `n` = 4:

```
n = input( 'Number of terms? ' );
s = 0;

for k = 1:n
  s = s + 1 / (k ^ 2);
end;

disp(sqrt(6 * s))
```

If you run this script for larger and larger values of `n` you will find that the output approaches a well-known limit. Can you figure out what it is?

(b) Rewrite the script using vectors and array operations.

2.20 Work through the following script by hand. Draw up a table of the values of `i`, `j` and `m` to show how their values change while the script executes. Check your answers by running the script.

```
v = [3 1 5];
i = 1;
for j = v
    i = i + 1;
    if i == 3
        i = i + 2;
        m = i + j;
    end
end
```

2.21 The steady-state current I flowing in a circuit that contains a resistance $R = 5$, capacitance $C = 10$, and inductance $L = 4$ in series is given by

$$I = \frac{E}{\sqrt{R^2 + (2\pi\omega L - \frac{1}{2\pi\omega C})^2}}$$

where $E = 2$ and $\omega = 2$ are the input voltage and angular frequency respectively. Compute the value of I. (Answer: 0.0396)

2.21 The electricity accounts of residents in a very small town are calculated as follows:

- if 500 units or less are used the cost is 2 cents per unit;
- if more than 500, but not more than 1000 units are used, the cost is $10 for the first 500 units, and then 5 cents for every unit in excess of 500;
- if more than 1000 units are used, the cost is $35 for the first 1000 units plus 10 cents for every unit in excess of 1000;
- in addition, a basic service fee of $5 is charged, no matter how much electricity is used.

Write a program which enters the following five consumptions into a vector, and uses a `for` loop to calculate and display the total charge for each one: 200, 500, 700, 1000, 1500. (Answers: $9, $15, $25, $40, $90)

2.23 Suppose you deposit \$50 per month in a bank account every month for a year. Every month, after the deposit has been made, interest at the rate of 1 percent is added to the balance, e.g. after one month, the balance is \$50.50, and after two months it is \$101.51. Write a program to compute and print the balance each month for a year. Arrange the output to look something like this:

```
MONTH    MONTH-END BALANCE

  1             50.50
  2            101.51
  3            153.02
 ...
 12            640.47
```

2.24 If you invest \$1000 for one year at an interest rate of 12 percent, the return is \$1120 at the end of the year. But if interest is compounded at the rate of 1 percent *monthly* (i.e. 1/12 of the annual rate), you get slightly more interest because it is compounded. Write a program which uses a `for` loop to compute the balance after a year of compounding interest in this way. The answer should be \$1126.83. Evaluate the formula for this result separately as a check: 1000×1.01^{12}.

2.25 A plumber opens a savings account with \$100 000 at the beginning of January. He then makes a deposit of \$1000 at the end of each month for the next 12 months (starting at the end of January). Interest is calculated and added to his account at the end of each month (before the \$1000 deposit is made). The monthly interest rate depends on the amount A in his account at the time when interest is calculated, in the following way:

$$\begin{aligned} A \leq 110\,000: &\quad 1 \text{ percent} \\ 110\,000 < A \leq 125\,000: &\quad 1.5 \text{ percent} \\ A > 125\,000: &\quad 2 \text{ percent} \end{aligned}$$

Write a program which displays, for each of the 12 months, under suitable headings, the situation at the end of the month as follows: the number of the month, the interest rate, the amount of interest and the new balance. (Answer: values in the last row of output should be 12, 0.02, 2534.58, 130263.78.)

2.26 It has been suggested that the population of the United States may be modeled by the formula

$$P(t) = \frac{19\,72\,73\,000}{1 + e^{-0.03134(t-1913.25)}}$$

where t is the date in years. Write a program to compute and display the population every *ten* years from 1790 to 2000. Try to plot a graph of the population against

time as well (Figure 7.14, accompanying Exercise 7.1 in Chapter 7, shows this graph compared with actual data).

Use your program to find out if the population ever reaches a 'steady state', i.e. whether it stops changing.

2.27 A mortgage bond (loan) of amount L is obtained to buy a house. The interest rate r is 15 percent (0.15) p.a. The fixed monthly payment P which will pay off the bond exactly over N years is given by the formula

$$P = \frac{rL(1 + r/12)^{12N}}{12[(1 + r/12)^{12N} - 1]}.$$

(a) Write a program to compute and print P if $N = 20$ years, and the bond is for \$50 000. You should get \$658,39.

(b) It's interesting to see how the payment P changes with the period N over which you pay the loan. Run the program for different values of N (use `input`). See if you can find a value of N for which the payment is less than \$625.

(c) Now go back to having N fixed at 20 years, and examine the effect of different interest rates. You should see that raising the interest rate by 1 percent (0.01) increases the monthly payment by about \$37.

2.28 It's useful to be able to work out how the period of a bond repayment changes if you increase or decrease your monthly payment P. The formula for the number of years N to repay the loan is given by

$$N = \frac{\ln\left(\frac{P}{P - rL/12}\right)}{12 \ln(1 + r/12)}.$$

(a) Write a new program to compute this formula. Use the built-in function `log` for the natural logarithm ln. How long will it take to pay off the loan of \$50 000 at \$800 a month if the interest remains at 15 percent? (Answer: 10.2 years—nearly twice as fast as when paying \$658 a month!)

(b) Use your program to find out by trial-and-error the smallest monthly payment that can be made to pay the loan off—this side of eternity. **Hint:** Recall that it is not possible to find the logarithm of a negative number, so P must not be less than $rL/12$.

3

Program design and algorithm development

The objectives of this chapter are to introduce you to the concept of program design and to the concept of a structure plan (pseudocode) as a means of designing the logic of a program.

This chapter provides an introduction to the design of computer programs. The top-down design process is elaborated to help you think about the development of good problem-solving strategies as they relate to the design of procedures to use software like MATLAB. We will consider the design of your own toolbox to be included among the toolboxes already available with your version of MATLAB, e.g. SIMULINK, the Symbolics toolbox, and the Controls toolbox. This is a big advantage of MATLAB (and tools like this software); it allows you to customize your working environment to meet your own needs. It is not only the 'mathematics handbook' of today's student, engineer and scientist, it is also a useful environment to develop software tools that go beyond any handbook to help you to solve relatively complicated mathematical problems. Your ability to use the complete power of MATLAB is limited only by your experience and educational background. The further you grow in your knowledge, you will more productively use your imagination and creative abilities to develop methods that tap the depths of this tool to help you solve technical problems. The problems could be problems assigned in course work, in undergraduate research, engineering and design projects, as well as (after graduation) on the job.

In the first part of this chapter we discuss the design process. In the second part of this chapter we will examine additional examples of the structure plan, which is the detailed description of the algorithm to be implemented. We will consider relatively simple programs; however, the process described is intended to provide insight into what you will confront when you deal with more complex problems. These are, of course, the engineering, scientific and

mathematical problems you learn during the later years of your formal education, your lifelong learning pursuits after graduation, and your continuing education responsibilities as required by your profession.

To be sure, the examples we have examined so far have been very simple logically. This is because we have been concentrating on the technical aspects of writing MATLAB statements correctly. It is very important to learn how MATLAB does the arithmetic operations that form the basis of more complex computer programs designed to compute numerical information. To design a successful program you need to understand a problem thoroughly, and to break it down into its most fundamental logical stages. In other words, you have to develop a systematic procedure or *algorithm* for solving the problem.

There are a number of methods which may assist in this process of algorithm development. In this chapter we look at one such approach, the *structure plan*, within the context of solving technical problems with the aid of a computer. You already have met briefly the concept of a structure plan. The development of this plan is the primary part of the software (or computer code) design process because it is the steps in this plan that are translated into a language the computer can understand, e.g. into MATLAB commands, some of which have already been introduced in the previous two chapters.

3.1 Computer program design process

The designs of useful utilities translated into the MATLAB language (either sequences of command lines or *functions*, which are described later in the text) and saved as M-files in your `work` directory are the primary and most useful goals of users of tools like MATLAB. There are numerous toolboxes available through MathWorks (among other sources) on a variety of engineering and scientific topics for individuals, industry and government agencies to purchase to help them do their work. A great example is the Aerospace Toolbox, which provides reference standards, environmental models, and aerodynamic coefficient importing for performing advanced aerospace analysis to develop and evaluate aerospace engineering designs. It is useful for you to search the MathWorks web site for the variety of products available to help the engineer, scientist and technical manager; it is located on the network at http://www.mathworks.com/.

In the default working directory, `\work`, or in your own working directory, e.g. `\mytools`, you will certainly begin to accumulate a set of M-files that you have

created as you continue to use MATLAB. One way to create and to get to your own working directory is to execute the following commands:

```
>> mkdir mytools   <Enter>
>> cd mytools   <Enter>
```

Certainly, you want to be sure that the tools you do save are reasonably well written (i.e. reasonably well designed). What does it mean to create well-written programs?

The goals of software tools design are that it works, it can easily be read and understood, and, hence, it can be systematically modified when required. For programs to work well they must satisfy the requirements associated with the problem or class of problems it is intended to solve. The specifications, i.e. the detailed description of the purpose (or function of the program), its inputs, method of processing, outputs and any other special requirements, must be known to design an effective algorithm and computer program. It must work completely and correctly, i.e. all options should be usable without error within the limitations of the specifications. The program must be readable and, hence, clearly understandable. Thus, it is useful to decompose major tasks (or the main program) into subtasks (or subprograms) that do specific parts of the main task. It is much easier to read subprograms that have fewer lines, than one large main program that doesn't segregate the subtasks effectively, particularly if the problem to be solved is relatively complicated. Each subtask should be designed so that it can be evaluated independently before it is implemented into the larger scheme of things (i.e. into the main program plan). A well-written code, when it works, is much more easily evaluated in the testing phase of the design process. If changes are necessary to correct sign mistakes and the like, they can be implemented easily. One thing to keep in mind when you add comments to describe the process programmed is this: add enough comments and references so that a year from the time you write the program you know exactly what was done and for what purpose. Note that the first few comment lines in a script file are displayed in the command window when you type `help` followed by the name of your file (naming your files is also an art).

The design process[1] is outlined next. The steps may be listed as follows:

Step 1: *Problem analysis.* The context of the proposed investigation must be established to provide the proper motivation for the design of a computer program. The designer must fully recognize the need and must develop an understanding of the nature of the problem to be solved.

[1] For a more detailed description of software design technology see, e.g. the book *C++ Data Structures* by Jones and Bartlett Publishers, Inc. in 1998.

Step 2: *Problem statement.* Develop a detailed statement of the mathematical problem to be solved with a computer program.

Step 3: *Processing scheme.* Define the inputs required and the outputs to be produced by the program.

Step 4: *Algorithm.* Design the step-by-step procedure using the *top-down* design process that decomposes the overall problem into subordinate problems. The subtasks to solve the latter are refined by designing an itemized list of steps to be programmed. This list of tasks is the *structure plan*; it is written in *pseudocode*, i.e. a combination of English, mathematics and anticipated MATLAB commands. The goal is to design a plan that is understandable and easily translated into a computer language.

Step 5: *Program algorithm.* Translate or convert the algorithm into a computer language (e.g. MATLAB) and debug the syntax errors until the tool executes successfully.

Step 6: *Evaluation.* Test all of the options and conduct a validation study of the computer program, e.g. compare results with other programs that do similar tasks, compare with experimental data if appropriate, and compare with theoretical predictions based on theoretical methodology related to the problems to be solved by the computer program. The objective is to determine that the subtasks and the overall program are correct and accurate. The additional debugging in this step is to find and correct *logical* errors (e.g. mistyping of expressions by putting a $+$ sign where a $-$ sign was supposed to be placed), and *run-time* errors that may occur after the program successfully executes (e.g. cases where division by `0` unintentionally occurs).

Step 7: *Application.* Solve the problems the program was designed to solve. If the program is well designed and useful it could be saved in your work directory (i.e. in your user-developed toolbox) for future use.

3.1.1 Projectile problem example

Step 1: Let us consider the projectile problem examined in first semester physics. It is assumed that the engineering and science students understand this problem (if it is not familiar to you, find a physics text that describes it or search the web; the formulae that apply to this problem will be provided in Step 2 of the design process). In this example we want to calculate the flight of a projectile (e.g. a golf ball) launched at a prescribed speed and a prescribed launch angle. We want to determine the trajectory of the flight path and the horizontal distance the projectile (or object) travels before it hits the ground. Let us assume zero air resistance and a constant gravitational force acting on the object in the opposite direction of the vertical distance from the ground. The launch angle,

θ_o, is defined as the angle measured from the horizontal (ground plane) upward towards the vertical direction, i.e. $0 < \theta_o \leq \pi/2$, where $\theta_o = 0$ implies a launch in the horizontal direction and $\theta_o = \pi/2$ implies a launch in the vertical direction (i.e. in the opposite direction of gravity). If $g = 9.81\,\text{m/s}^2$ is used as the acceleration of gravity, then the launch speed, V_o, must be entered in units of m/s. Thus, if the time, $t > 0$, is the time in seconds (s) from the launch time of $t = 0$, the distance traveled in x (the horizontal direction) and y (the vertical direction) are in meters (m). We want to determine the time it takes from the commencement of motion to hit the ground, the horizontal distance traveled and the shape of the trajectory. In addition, plot the speed of the projectile versus the direction of this vector. We need, of course, the theory (or mathematical expressions) that describes the solution to this problem in order to develop an algorithm to obtain solutions to the zero-resistance projectile problem.

Step 2: The mathematical formulae that describe the solution to the projectile problem described in Step 1 are provided in this step. Given the launch angle and launch speed, the horizontal distance traveled from $x = y = 0$, which is the coordinate location of the launcher, is

$$x_{max} = 2\frac{V_o^2}{g}\sin\theta_o\cos\theta_o.$$

The time from $t = 0$ at launch for the projectile to reach x_{max} (i.e. its range) is

$$t_{x_{max}} = 2\frac{V_o}{g}\sin\theta_o.$$

The object reaches its maximum altitude,

$$y_{max} = \frac{V_o^2}{2g}\sin^2\theta_o,$$

at time

$$t_{y_{max}} = \frac{V_o}{g}\sin\theta_o.$$

For this problem the horizontal distance traveled when the object reaches the maximum altitude is $x_{y_{max}} = x_{max}/2.0$.

The trajectory (or flight path) is described by the following pair of coordinates at a given instant of time between $t = 0$ and $t_{x_{max}}$:

$$x = V_o t\cos\theta_o,$$

$$y = V_o t\sin\theta_o - \frac{g}{2}t^2.$$

We need to solve these equations over the range of time $0 < t \leq t_{x_{max}}$ for prescribed launch conditions $V_o > 0$ and $0 < \theta \leq \pi/2$. Then the maximum value of the altitude and the range are computed along with their respective arrival times. Finally, we want to plot V versus θ, where

$$V = \sqrt{(V_o \cos\theta_o)^2 + (V_o \sin\theta_o - gt)^2},$$

and

$$\theta = \tan^{-1}\left(\frac{V_y}{V_x}\right).$$

We must keep in mind when we study the solutions based on these formulae that the air resistance was assumed negligible and the gravitational acceleration was assumed constant.

Step 3: The required inputs are g, V_o, θ, and a finite number of time steps between $t = 0$ and the time the object returns to the ground. The outputs are the range and time of flight, the maximum altitude and the time it is reached, and the shape of the trajectory in graphical form.

Steps 4 & 5: The structure plan developed to solve this problem is given next as a MATLAB program; this is because it is a realitively straightforward problem and the translation to MATLAB is well commented with details of the approach applied to solve this problem, i.e. the steps of the structure plan are enumerated. This plan and M-file, of course, is the summary of the results developed by trying a number of approaches during the design process and, thus, discarding numerous sheets of scratch paper before summarizing the results! There are more explicit examples of structure plans for your review and investigation in the next section of this chapter. You must keep in mind that it was not difficult to enumerate a list of steps associated with the general design process, i.e. the technical problem-solving methodology. However, it is certainly not so easy to implement the steps because the steps draw heavily from your design-of-technical solutions experience. Hence, we must begin by studying the design of relatively simple programs like the one described in this section. The evaluated and tested code is as follows:

```
%
% The proctile problem with zero air resistance
% in a gravitational field with constant g.
%
% Written by D. T. Valentine ........ september 2006
% An eight-step structure plan applied in MATLAB:
%
% 1. Definition of the input variables.
```

```
%
g = 9.81; % Gravity in m/s/s.
vo = input('What is the launch speed in m/s?')
tho = input('What is the launch angle in degrees?')
tho = pi*tho/180; % Conversion of degrees to radians.
%
% 2. Calculate the range and duration of the flight.
%
txmax = (2*vo/g) * sin(tho);
xmax = txmax * vo * cos(tho);
%
% 3. Calculate the sequence of time steps to compute
     trajectory.
%
dt = txmax/100;
t = 0:dt:txmax;
%
% 4. Compute the trajectory.
%
x = (vo * cos(tho)) .* t;
y = (vo * sin(tho)) .* t - (g/2) .* t.^2;
%
% 5. Compute the speed and angular direction of the
     projectile.
% Note that vx = dx/dt, vy = dy/dt.
%
vx = vo * cos(tho);
vy = vo * sin(tho) - g .* t;
v = sqrt(vx.*vx + vy.*vy);
        th = (180/pi) .* atan2(vy,vx);
%
% 6. Compute the time, horizontal distance at maximum
     altitude.
%
tymax = (vo/g) * sin(tho);
        xymax = xmax/2;
ymax = (vo/2) * tymax * sin(tho);
%
% 7. Display ouput.
%
disp(['Range in m = ',num2str(xmax),' Duration in s = ',
       num2str(txmax)])
       disp(' ')
disp(['Maximum altitude in m = ',num2str(ymax),' Arrival
       in s = ', num2str(tymax)])
plot(x,y,'k',xmax,y(size(t)),'o',xmax/2,ymax,'o')
title(['Projectile flight path, vo =',num2str(vo),'
```

```
     th =', num2str(180*th/pi)])
xlabel('x'), ylabel('y')   % Plot of Figure 1.
figure % Creates a new figure.
plot(v,th,'r')
title('Projectile speed vs. angle')
xlabel('V'), ylabel('\theta') % Plot of Figure 2.
%
% 8. Stop.
%
```

Steps 6 & 7: The program was evaluated by executing a number of values of the launch angle and launch speed within the required specifications. The angle of 45 degrees was checked to determine that the maximum range occurred at this angle for all specified launch speeds. This is well known for the zero air resistance case in a constant g force field. Executing this computer code for $V_o = 10$ m/s and $\theta_o = 45$ degrees, the trajectory and the plot of orientation versus speed in Figures 3.1 and 3.2, respectively, were produced.

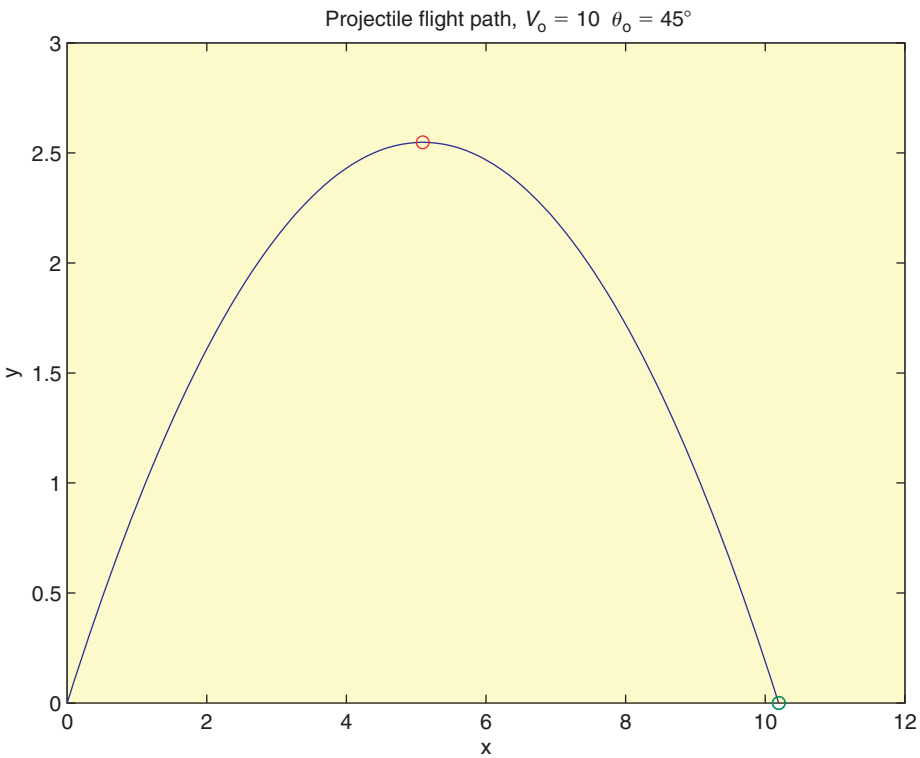


Figure 3.1 Projectile trajectory

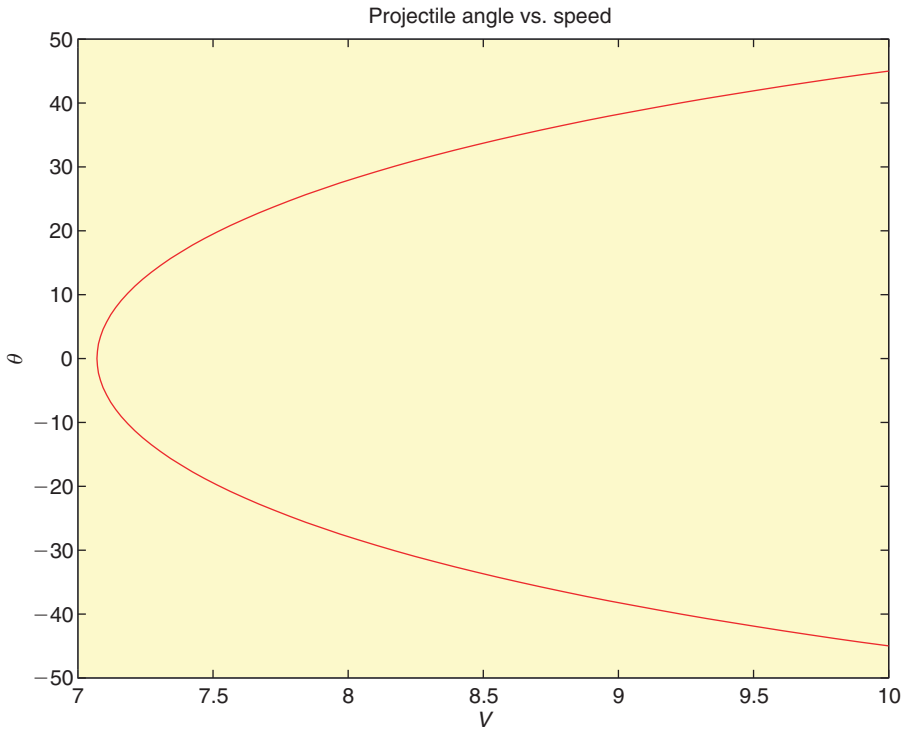


Figure 3.2 Projectile angle vs. speed

How can you find additional examples of MATLAB programs (good ones or otherwise) to help develop tools to solve your own problems? We all recognize that learning by example isn't a bad way of learning how to use tools. New tools are continually being developed by the users of MATLAB. As a new tool proves to be of more general use, MathWorks may include it in their list of products (if, of course, the author of a new tool desires to do this). There are also many examples of useful scripts that are placed on the world-wide-web (www) for anyone interested in them. They, of course, must be evaluated carefully since it is the users' responsibility and *not* the creators' responsibility for the correctness of the results of the tool applied. This responsibility holds for all tools applied by the engineer and the scientist. Hence, it is very important that (just as in using a laboratory apparatus) the engineer or the scientist prove to themselves that the tool they are using is indeed valid for the problem they are trying to solve.

To illustrate how easy it is to find examples of scripts the author typed `MATLAB examples` in one of the available search engines and found the following script (among many others):

```
>> t = (0:.1:2*pi)';
>> subplot(2,2,1)
```

```
>> plot(t,sin(t))
>> subplot(2,2,2)
>> plot(t,cos(t))
>> subplot(2,2,3)
>> plot(t,exp(t))
>> subplot(2,2,4)
>> plot(t,1./(1+t.^2))
```

This script illustrates how to put four plots in a single figure window. To check that it works, type each line in the command window followed by Enter. Note the positions of each graphic; its location is determined by the three integers in the `subplot` function list of arguments. Search the `Help` via the question mark (`?`) for more information on `subplot`.

The next example is provided to illustrate how to find more advanced tools from appropriate sources available on the world-wide-web (www). It is obviously not so easy to read if you are not familiar with the dynamical system investigated by its author. Hence, a comment or two on finding information related to this problem (as an example of doing research) are also provided for you to consider. From the web site http://www.mathworks.com/matlabcentral/files/12173/mandelbrot14.m, which is at 'MATLAB Central', an open exchange for the MATLAB user community, the author downloaded (by 'cut-and-paste') the following content of the M-file `mandelbrot14.m`. The author of this file and other notes are given at the top of the file. You can either type this *function* file into the Editor and save it under its given name or download it directly from the web site into your working directory. It is important that you name the file `mandelbrot14.m` because it is a `function` file. Then you can execute it to see what it does; to execute it you need to type the following command: `>>mandelbrot14(1)`. Don't forget to hit Enter! It took a couple of minutes to execute on a fairly fast computer; hence, have a little patience waiting for the graph of the Mandelbrot set to appear.

```
function mandelbrot14(any)
% Mandelbrot set v0.14
% Sai Wing Man 05/09/2006
% RUN: 'mandelbrot14(1)' first to get the axes, then
   'mandelbrot14' (no argin) after..
% .. but first zooming in using the zoom tool.
% I don't ever intend to update this file.
%{
Copyright (C) 2006 Sai Wing Man
```

```
This program is free software; you can redistribute it
and/or modify it under the terms of the GNU General
Public License as published by the Free Software
Foundation; either version 2 of the License, or (at your
option) any later version.

This program is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the
implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE. See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public
License along with this program; if not, write to the
Free Software Foundation, Inc., 51 Franklin Street,
Fifth Floor, Boston, MA 02110-1301 USA.
%}
tic
if nargin==1
%default axes
    x0=-2;x1=0.5;y0=-1;y1=1;
    axis([x0 x1 y0 y1])
end
k=axis; x0=k(1); x1=k(2); y0=k(3); y1=k(4);
clc
clf
num=440;
[x,y]=meshgrid(x0:(x1-x0)/(num-1):x1, (y0:(y1-y0)/
   (num-1):y1)*i);
c=x+y;
z=zeros(num,num);
jvec=5:30:500;
jindex=1;
cmap=colormap(hsv(length(jvec)));
hold on
for j=1:max(jvec)
    z=z.^2+c;
    if jvec(jindex)==j
        if jindex<length(jvec)
        plot(real(~isnan(z).*c),imag(~isnan(z).*
        c),'.', ...
                        'Markersize',0.5,'Color',cmap
                           (jindex,:))
        end
        clc
        jindex=jindex+1;
```

```
        perc=j/max(jvec)*100/2+(jindex-2)/length(jvec)
            *100/2;
        disp([int2str(perc)'% done'])
    end
end
plot(real(~isnan(z).*c),imag(~isnan(z).*c),'.',
'Markersize',0.5,'Color',[0 0 0])
axis([x0 x1 y0 y1])
clc
toc
```

This sample was downloaded from the www to illustrate the fact that relatively complex problems have already been programmed by others. Assuming that you typed and named this file correctly, you will get a quite beautiful graph. This sample illustrates the results of the following sort of search for information: If you were interested in some of the mathematical contributions of Benoit Mandelbrot, then you may find examples of solved problems related to his work on the web. Such examples may help you in your investigation of the consequences of his contributions particularly when you need to generate programs to examine his mathematical ideas. The M-file `madelbrot14.m` is an example of how to present solutions of a nonlinear problem developed by Sai Wing Man, version 14 (2006). There are a couple of key command lines in the code. One is `c=x+y;`, where `x` and `y` are coordinates of points on a grid in the complex plane generated by the `meshgrid` function. The other key command is `z=z.^2+c;`. A color map is created depending on the value of `z` each time through the `for` loops. This map is an illustration of the Mandelbrot set as described in Mandelbrot's book entitled the *The Fractal Geometry of Nature* published by the W. H. Freeman Co., NY (1983). The implications of the graphical illustration generated by this tool are beyond the scope of this presentation; however, it is an interesting illustration of how to generate a beautiful picture with mathematics. Also, search your library for more information on this fascinating topic. This topic and other topics in nonlinear phenomena are becoming more and more important in engineering and in the sciences. Tools like MATLAB can help us learn more about these topics when we confront them later on in our lifelong continuing education.

This type of search can be done on any topic of current interest to the academic, research, development and design communities. If you are good at searching the world-wide-web, you certainly can benefit from the information available on it; the skill of searching is a skill that is learned by practice and enhanced by need. The skill of finding helpful information on technical matters and the skill associated with using tools like MATLAB require practice! There are numerous

contributions by others to the 'computer-program literature' like this one for us to sample.

3.2 Other examples of structure plans

A structure plan is typically written in what is called *pseudo-code*, i.e. statements in English, mathematics and MATLAB, describing in detail how to solve a problem. You don't have to be an expert in any particular computer language to understand pseudocode. A structure plan may be written at a number of levels, each of increasing complexity, as the logical structure of the program is developed.

Suppose we want to write a script to convert a temperature on the Fahrenheit scale (where water freezes and boils at 32° and 212°, respectively) to the more familiar Celsius centigrade scale. A first-level structure plan might be a simple statement of the problem:

1. Initialize Fahrenheit temperature
2. Calculate and display Celsius temperature
3. Stop.

Step 1 is pretty straightforward, but step 2 needs elaborating, so the second-level plan could be something like this:

1. Initialize Fahrenheit temperature (*F*)
2. Calculate Celsius temperature (*C*) as follows:
   2.1. Subtract 32 from *F* and multiply by 5/9
3. Display the value of *C*
4. Stop.

There are no hard and fast rules about how to write structure plans. The essential point is to cultivate the mental discipline of getting the logic of a problem clear before attempting to write the program. The 'top-down' approach of structure plans means that the overall structure of a program is clearly thought out before you have to worry about the details of syntax (coding), and this reduces the number of errors enormously.

1. Start
2. Input data (a, b, c)
3. If $a = 0$ then
   If $b = 0$ then
      If $c = 0$ then
         Display 'Solution indeterminate'
      else
         Display 'There is no solution'
   else
      $x = -c/b$
      Display x (only one root: equation is linear)
   else if $b^2 < 4ac$ then
      Display 'Complex roots'
   else if $b^2 = 4ac$ then
      $x = -b/(2a)$
      Display x (equal roots)
   else
      $x_1 = (-b + \sqrt{b^2 - 4ac})/(2a)$
      $x_2 = (-b - \sqrt{b^2 - 4ac})/(2a)$
      Display x_1, x_2
4. Stop.

Figure 3.3 Quadratic equation structure plan

A script to implement this is as follows:

```
% Script to convert temperatures from F to C
% by D.T.V.........October 2006
%
F = input('Temperature in degrees F:')
          C = (F-32)*5/9;
disp(['Temperature in degrees C = ',num2str(C)])
% STOP
```

Two checks of the tool were done. They were for $F = 32$, which gave $C = 0$, and $F = 212$, which gave $C = 100$. The results were found to be correct and, hence, this simple script is, as such, validated.

3.2.1 Quadratic equation

When you were at school you probably solved hundreds of quadratic equations of the form

$$ax^2 + bx + c = 0.$$

A structure plan of the *complete* algorithm for finding the solution(s) x, given any values of a, b and c, is shown in Figure 3.3. Figure 3.4 shows the graph of a quadratic equation with real unequal roots.

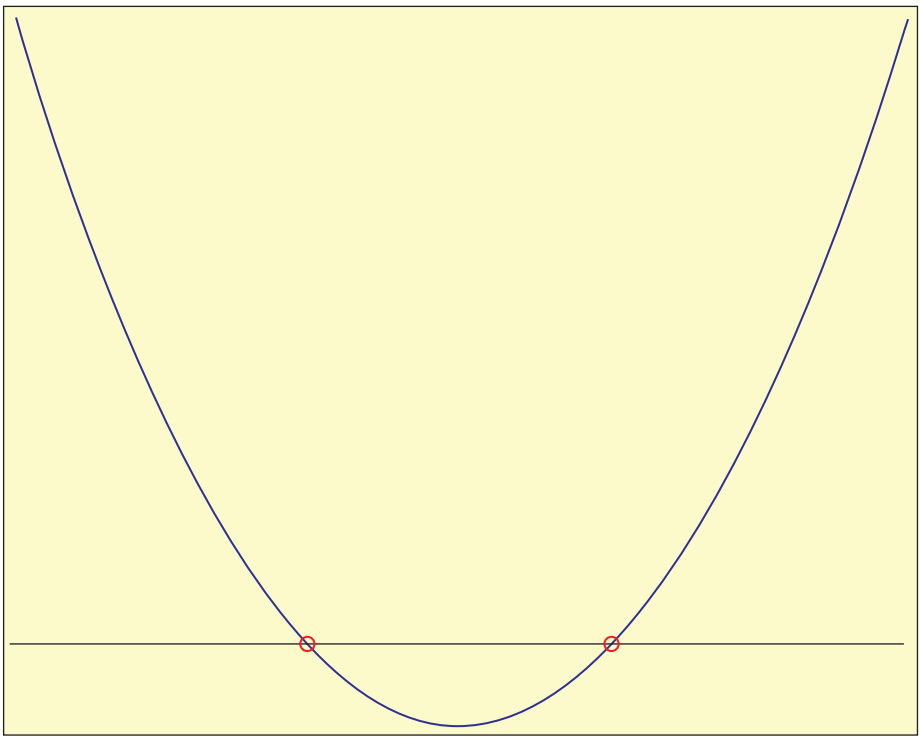


Figure 3.4 Graph of a quadratic equation with real unequal roots indicated by ○

When you come to writing the program, you can type the structure plan into the MATLAB Editor, as described in Chapter 2. Use cut and paste to make another copy of the plan below the first one, and translate the second copy into MATLAB statements. If it's a good structure plan, it should translate line for line. Then comment out the original structure plan with **Text -> Comment**, so you can save it with the program.

3.3 Structured programming with functions

Many examples later in this book will get rather involved. More advanced programs like these should be *structured* by means of your own *function M-files*. These are dealt with in detail in Chapter 10. A function M-file is a type of script file (i.e. a file with an extension of `.m`) which you can 'call' interactively, or from other scripts, in specific ways.

The 'main' script itself will then look very much like a first-level structure plan of the problem.

For example, the quadratic equation problem may be structure-planned at the first level as follows:

1. Input the data
2. Find and display the solution(s)
3. Stop.

Using a function file `quad.m` to do the dirty work, this could be translated directly into the following MATLAB script:

```
a = input('Enter coefficients: ');
x = quad(a);
```

(The details of how to code this particular problem are left as an exercise in Chapter 10.)

Summary

- An algorithm is a systematic logical procedure for solving a problem.
- An algorithm must be developed for a problem before it can be coded.
- A structure plan is a representation of an algorithm in pseudocode.
- A function M-file is a type of script file designed to handle a particular task, and which may be activated (invoked) whenever needed.

EXERCISES

The problems in these exercises should all be structure-planned, before being written up as MATLAB programs (where appropriate).

3.1 The structure plan in this example defines a geometric construction. Carry out the plan by sketching the construction:

1. Draw two perpendicular *x*- and *y*-axes
2. Draw the points *A* (10, 0) and *B* (0, 1)
3. While *A* does not coincide with the origin repeat:
   Draw a straight line joining *A* and *B*
   Move *A* one unit to the left along the *x*-axis
   Move *B* one unit up on the *y*-axis
4. Stop.

3.2 Consider the following structure plan, where M and N represent MATLAB variables:

1. Set $M = 44$ and $N = 28$
2. While M not equal to N repeat:
   While $M > N$ repeat:
   Replace value of M by $M - N$
   While $N > M$ repeat:
   Replace value of N by $N - M$
3. Display M
4. Stop.

(a) Work through the structure plan, sketching the contents of M and N during execution. Give the output.
(b) Repeat (a) for $M = 14$ and $N = 24$.
(c) What general arithmetic procedure does the algorithm carry out (try more values of M and N if necessary)?

3.3 Write a program to convert a Fahrenheit temperature to Celsius. Test it on the data in Exercise 2.11 (where the reverse conversion is done).

3.4 Write a script which inputs any two numbers (which may be equal) and displays the larger one with a suitable message, or if they are equal, displays a message to that effect.

3.5 Write a script for the general solution of the quadratic equation $ax^2 + bx + c = 0$. Use the structure plan in Figure 3.3. Your script should be able to handle all possible values of the data a, b, and c. Try it out on the following values of a, b and c:

(a) 1, 1, 1 (complex roots);
(b) 2, 4, 2 (equal roots of -1.0);
(c) 2, 2, -12 (roots of 2.0 and -3.0).

The structure plan in Figure 3.3 is for programming languages which cannot handle complex numbers. MATLAB however can. Adjust your script so that it can also find complex roots. Test it on case (a) above; the roots are $-0.5 \pm 0.866i$.

3.6 Develop a structure plan for the solution of two simultaneous linear equations (i.e. the equations of two straight lines). Your algorithm must be able to handle all possible situations, viz. lines which are intersecting, parallel, or coincident. Write a program to implement your algorithm, and test it on some equations for which you know the solutions, e.g.

$$\begin{aligned} x + y &= 3 \\ 2x - y &= 3 \end{aligned}$$

($x = 2$, $y = 1$). **Hint**: Begin by deriving an algebraic formula for the solution of the system

$$ax + by = c$$
$$dx + ey = f.$$

The program should input the coefficients a, b, c, d, e and f.

We will see in Chapter 16 that MATLAB has a very elegant way of solving systems of equations directly, using matrix algebra. But it is good for the development of your programming skills to do it the long way, as in this exercise.

4

MATLAB functions & *data import-export utilities

The objectives of this chapter are:

(1) **To enable you to become familiar with some of the more common MATLAB functions.**

(2) **To introduce you briefly to ways of importing and exporting data in and out of the MATLAB workspace using:**

- **The `load` and `save` commands.**
- **The Import Wizard.**
- **The low-level file input/output (I/O) functions.**

So far you should be able to write a MATLAB program which inputs data, performs simple arithmetic operations on the data, perhaps involving loops and decisions, and displays the results of the computation in a comprehensible form. However, more interesting problems in science and engineering are likely to involve special mathematical functions like sines, cosines, logarithms, etc. MATLAB comes with a large collection of such functions; we have seen some of them already. This chapter introduces you to the more common functions available in MATLAB. In addition, you may wish to import data to be plotted or operated on mathematically, and export data for future use. Hence, this chapter also introduces you to the importing of data into the MATLAB workspace from various sources. It also discusses the exporting of data to files in your `work` directory for later use with MATLAB or with other software tools.

4.1 Some common functions

Tables of MATLAB functions and commands appear in Appendix C. A short list of some of the more common ones follows. Use `helpwin` at the command line to see a list of categories of functions, with links to descriptions of the functions. Alternatively, go to the **Contents** listing in the Help Navigator (the left pane in the Help browser), and expand successively MATLAB, **Reference**, **MATLAB Function Reference** where you can choose either **Functions by Category**, or **Alphabetical List of Functions**.

Note that if the argument of a function is an array, the function is applied element by element to all the values in the array, e.g.

```
sqrt([1 2 3 4])
```

returns

```
1.0000    1.4142    1.7321    2.0000
```

Since this text is written in a tutorial style, it is expected that you will examine the following list of some of the more common functions in MATLAB. It is also, to some extent, expected that you already know, from your first courses in mathematics and science, something about these functions. One way to examine these functions is to plot them. Have some fun experimenting with MATLAB in your investigation of the following functions! For example, do the following exercise for all functions of the assigned variable `x`, as illustrated next.

```
x = -1:.1:1; <Enter>
plot(x,abs(x),'o') <Enter>
```

You should get an illustration that looks like a V, *for victory of the successful implementation of this task*!

`abs(x)`
: absolute value of `x`.

`acos(x)`
: arc cosine (inverse cosine) of `x` between 0 and π.

`acosh(x)`
: inverse hyperbolic cosine of `x`, i.e. $\ln(x + \sqrt{x^2 - 1})$.

`asin(x)`
: arc sine (inverse sine) of `x` between $-\pi/2$ and $\pi/2$.

`asinh(x)`
: inverse hyperbolic sine of `x`, i.e. $\ln(x+\sqrt{x^2+1})$.

`atan(x)`
: arc tangent of `x` between $-\pi/2$ and $\pi/2$.

`atan2(y, x)`
: arc tangent of `y/x` between $-\pi$ and π.

`atanh(x)`
: inverse hyperbolic tangent of `x`, i.e. $\frac{1}{2}\ln\left(\frac{1+x}{1-x}\right)$.

`ceil(x)`
: smallest integer which exceeds `x`, i.e. rounds up to nearest integer, e.g. `ceil(-3.9)` returns `-3`, `ceil(3.9)` returns `4`.

`clock`
: time and date in a six-element vector, e.g. the statements

```
t = clock;
fprintf( ' %02.0f:%02.0f:%02.0f\n', t(4), t(5),
        t(6) );
```

result in `14:09:03`. Note how the hours, minutes and seconds are left-filled with zeros if necessary.

`cos(x)`
: cosine of `x`.

`cosh(x)`
: hyperbolic cosine of `x`, i.e. $\frac{e^x+e^{-x}}{2}$ (see Figure 4.1).

`cot(x)`
: cotangent of `x`.

`csc(x)`
: cosecant of `x`.

`cumsum(x)`
: cumulative sum of the elements of `x`, e.g. `cumsum(1:4)` returns `[1 3 6 10]`.

`date`
: date in a string in dd-mmm-yyyy format, e.g. `02-Feb-2001`.

`exp(x)`
: value of the exponential function e^x (see Figure 4.1).

`fix(x)`
: rounds to the nearest integer towards zero, e.g. `fix(-3.9)` returns `-3`, `fix(3.9)` returns `3`.

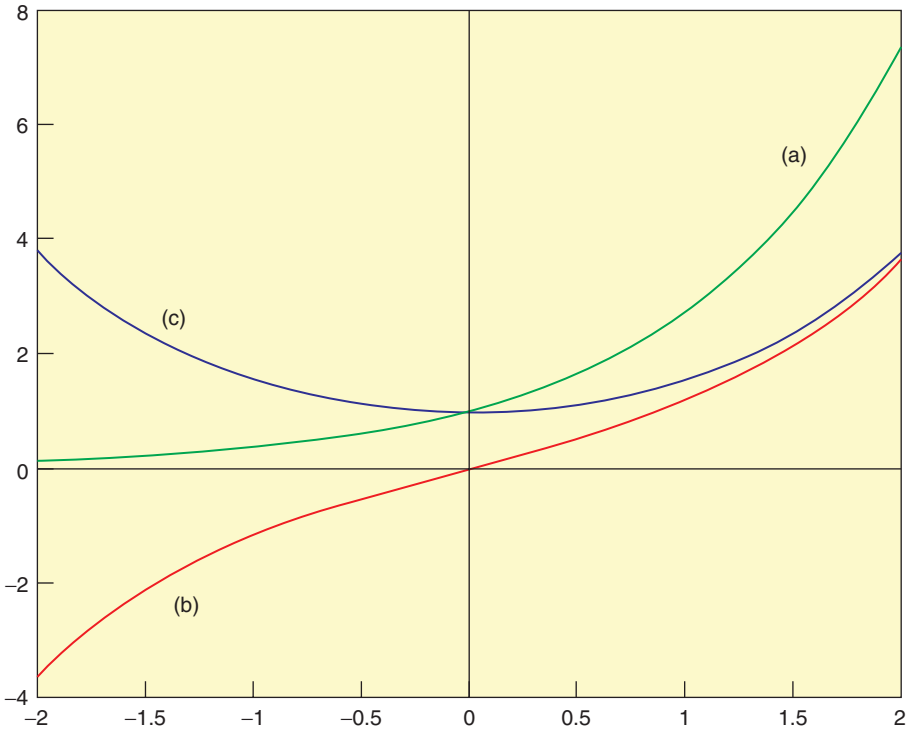


Figure 4.1 The (a) exponential, (b) hyperbolic sine and (c) hyperbolic cosine functions.

`floor(x)`
: largest integer not exceeding `x`, i.e. rounds down to nearest integer, e.g. `floor(-3.9)` returns `-4`, `floor(3.9)` returns `3`.

`length(x)`
: number of elements of vector `x`.

`log(x)`
: natural logarithm of `x`.

`log10(x)`
: base 10 logarithm of `x`.

`max(x)`
: maximum element of vector `x`.

`mean(x)`
: mean value of elements of vector `x`.

`min(x)`
: minimum element of vector `x`.

`pow2(x)`
: 2^x.

`prod(x)`
: product of the elements of `x`.

`rand`
: pseudo-random number in the interval [0, 1). The value returned is *pseudo*-random rather than truly random in the sense that there is an algorithm that determines `rand` from the initial 'seed'. The same seed will generate the same 'random' sequence; see a later chapter for how to seed `rand` by looking in the index.

`realmax`
: largest positive floating point number on your computer.

`realmin`
: smallest positive floating point number on your computer.

`rem(x, y)`
: remainder when `x` is divided by `y`, e.g. `rem(19, 5)` returns `4` (5 goes 3 times into 19, remainder 4).

  Strictly, `rem(x, y)` returns `x - y * n` where `n = fix(x/y)` is the integer nearest to `x/y`. This shows how negative and/or non-integer arguments are handled.

  `fix` and `rem` are useful for converting smaller units to larger ones, e.g. inches to feet and inches (one foot = 12 inches). The following statements convert 40 inches this way:

```
feet = fix(40/12)
inches = rem(40, 12)
```

Let us next examine an example that will hopefully inspire you to examine all of the functions listed as well as any other function that you may discover that MATLAB supports. Let us consider the arc-cosine, the arc-sine and the arc-tangent functions, i.e. `acos(x)`, `asin(x)` and `atan(x)`, respectively. If you specify `x`, i.e. the cosine, the sine and the tangent, respectively, between -1 and 1, what quadrant of the circle are the output angles selected? To provide an answer, the following *M-file* script was created and executed. The graphical comparison of the computed results is illustrated in Figure 4.2. The `REMARKS` at the end of the script provides an interpretation of the graphical results and, hence, an answer to the question raised.

```
%
% Script to compare the acos(x), asin(x), and atan(x)
% functions over the range -1 < x < 1. The values are
% converted to angles in degrees. The results are
% compared graphically.
%
```

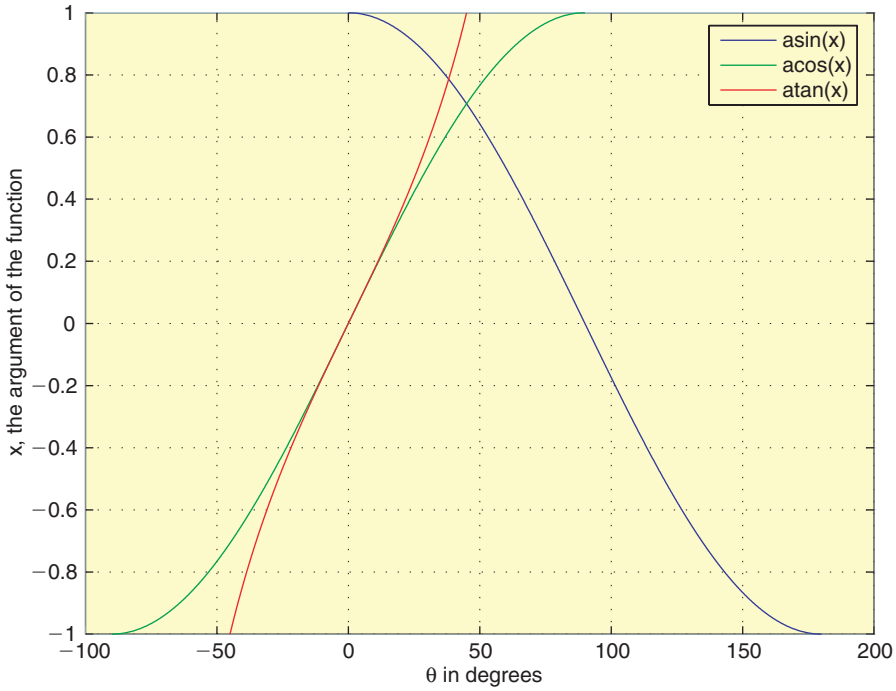


Figure 4.2 A comparison of the results from the acos, asin and atan functions.

```
% Script prepared by D. T. Valentine - September 2006.
%
% The question raised is: What range of angles, i.e.
% which of the four quadrants of the circle from 0 to
% 2*pi are the angular outputs of each of the functions?
%
% Determine the values of x to be examined:
%
x = -1:0.001:1;
%
% Compute the arc-functions:
%
y1 = acos(x);
y2 = asin(x);
y3 = atan(x);
%
% Convert the angles from radians to degrees:
%
y1 = 180*y1/pi;
y2 = 180*y2/pi;
y3 = 180*y3/pi;
%
```

```
% Plot the results:
%
plot(y1,x,y2,x,y3,x),grid,legend('asin(x)', 'acos(x)',
'atan(x)')
xlabel('\theta in degrees'),ylabel('x, the argument of
the function')
%
% REMARKS: Note the following:
%     (1) The acos(x) varies from 0 to 90 to 180 degrees.
%     (2) The asin(x) varies from -90 to 0 to 90 degrees.
%     (3) The atan(x) varies from -45 to 0 to 45 degrees.
%
% Stop
```

4.2 *Importing and exporting data

When you get into serious programming you will often need to store data on a disk. The process of moving data between MATLAB and disk files is called importing (from disk files) and exporting (to disk files). Data are saved in disk files in one of two formats: text or binary. In text format, data values are ASCII codes, and can be viewed in any text editor. In binary format, data values are not ASCII codes and cannot be viewed in a text editor. Binary format is more efficient in terms of storage space required. This chapter provides a brief summary of the main ways in which MATLAB imports and exports data. For full details consult **Using MATLAB: Development Environment: Importing and Exporting Data**.

4.2.1 The `load` and `save` commands

If you want to save data between MATLAB sessions the `save` and `load` commands are probably the best ones to use.

4.2.2 Exporting text (ASCII) data

To export (save) the array

```
A =
        1      2      3
        4      5      6
```

in 'delimited' ASCII format in the file `myData.txt` use the command

```
save myData.txt A -ascii
```

If you view `myData.txt` in a text editor (or `type` it in the Command Window) it looks like this:

```
1.0000000e+000  2.0000000e+000  3.0000000e+000
4.0000000e+000  5.0000000e+000  6.0000000e+000
```

Delimiters are the characters used to separate the data values in the file—spaces by default. You can use tabs instead of spaces by specifying the `-tabs` qualifier instead of `-ascii`. If you save character arrays (strings) in this way, the ASCII codes of the characters are written to the file.

4.2.3 Importing text (ASCII) data

The `load` command is the reverse of `save`, but has a curious twist. If the array `A` has been saved in `myData.txt` as above the command

```
load myData.txt
```

creates a variable in the workspace *with the same name as the file, minus the extension*, i.e. `myData`. If you don't want the filename as the variable name use the functional form of the command, e.g.

```
A = load('myData.txt')
```

Data imported in this way doesn't have to be created by MATLAB. You can create it in a text editor, or it could be created by any other program that exports data in ASCII format.

4.2.4 Exporting binary data

The command

`save` *filename* `x y z`

saves the variables `x`, `y` and `z` in the file *filename*`.mat` in MATLAB proprietary binary format, i.e. such a MAT-file can only be used by MATLAB.

Note:

- If no variables are listed the entire workspace is saved.
- The extension `.mat` is the default—you can specify a different extension.
- Seek `help` for all the `save` options.

Importing binary data

The command

`load` *filename*

loads all the variables from *filename*`.mat` into the workspace; see `help` for all the `load` options.

4.2.5 The Import Wizard

The MATLAB Import Wizard is the easiest way of importing data into the workspace during an interactive session.

Importing ASCII data

Use the Import Wizard to import ASCII data as follows:

- Start the Import Wizard by selecting **Import Data** on the MATLAB **File** menu. You can also use `uiimport` on the command line to start it. A list of files appears in a dialogue box. **Open** the file you want to import.
- Select the delimiter used in the text file (if necessary). Click **Next**.
- Select the variables you want to import. By default the Import Wizard puts all the numeric data in one variable and all the text data in other variables, but you can choose other options.
- Click **Finish** to import the data into the selected variables.

Importing binary data

To import binary data with the Import Wizard, start it in the same way as when importing text data. When the file opens the Import Wizard attempts to process its contents and creates variables depending on the type of data in the file. Check the variables you want to import and click **Finish** to create the selected

variables. You can, for example, import data from an Excel spreadsheet in this way. If the data are numeric with row and column headers, the Import Wizard imports the numeric data into a numeric array and the headers into a cell array. Neat isn't it?

4.2.6 Low-level file I/O functions

MATLAB has a set of low-level file I/O (input/output) functions based on the I/O functions of the ANSI Standard C Library. You would typically use these functions to access binary data written by C or Java programs, for example, or to access a database which is too large to be loaded into the workspace in its entirety.

C programmers should note that the MATLAB file I/O commands are not all identical to their C counterparts. For example, `fread` is 'vectorized', i.e. it reads until it encounters a text string or the end of file.

Files can be accessed with these functions in text or binary mode. In binary mode you can think of the file as a long continuous stream of bytes, which are your responsibility to interpret correctly. Files opened in binary mode can be accessed 'randomly', i.e. you can specify at which particular byte you want to start reading or writing.

The short programs which follow show how to create a file in binary mode, how to read it, and how to change it. Explanations of new features follow each program. You will need to consult the online documentation and `help` to see the wide range of options available for these I/O functions.

Writing binary data

The example we are going to use has a wide variety of applications. We want to set up a database of *records*, each record consisting of information about individuals (clients, customers, students). In this example each record will have a student's name and one mark. (The term 'record' has no particular significance in MATLAB, as it does, for example, in Pascal. It is used here as a convenient way of thinking of the basic unit in a database.)

The following program (`writer.m`) invites you to enter any number of names and marks from the command line, and writes them as binary data to a file. To terminate the process just hit **Enter** for the next name.

```
namelen = 10;                          % 10 bytes for name
fid = fopen('marks.bin', 'w');         % open for write only
str = '?';                             % not empty to start
```

```
while ~isempty(str)
    str = input( 'Enter name: ', 's' );
    if ~isempty(str)
        if length(str) > namelen
            name = str(1:namelen);    %only first ten
                                       chars allowed
        else
            name = str;
            name(length(str)+1:namelen) = ' '; %pad with
                                                blanks if
                                                ... too
                                                short
        end
        fwrite(fid, name);
        mark = input( 'Enter mark: ' );
        fwrite(fid, mark, 'float');             % 4 bytes
                                                  for mark
    end
end

fclose(fid);
```

Note:

- The statement

  ```
  fid = fopen('marks.bin', 'w');
  ```

  creates a file `marks.bin` for writing only. If the file is successfully opened `fopen` returns a non-negative integer called the *file identifier* (`fid`) which can be passed to other I/O functions to access the opened file. If `fopen` fails (e.g. if you try to open a non-existent file for reading) it returns `-1` to `fid` and assigns an error message to an optional second output argument.

  The second argument `'w'` of `fopen` is the permission string and specifies the kind of access to the file you require, e.g. `'r'` for read only, `'w'` for write only, `'r+'` for both reading and writing, etc. See `help fopen` for all the possible permission strings.

- The `while` loop continues asking for names until an empty string is entered (`str` must therefore be non-empty initially).

- Each name written to the file must be the same length (otherwise you won't know where each record begins when it comes to changing them). The `if` statement ensures that each name has exactly 10 characters (`namelen`)

no matter how many characters are entered (the number 10 is arbitrary of course!).

- The first `fwrite` statement

```
fwrite(fid, name);
```

writes all the characters in `name` to the file (one byte each).

- The second `fwrite` statement

```
fwrite(fid, mark, 'float');
```

writes `mark` to the file. The third (optional) argument (`precision`) specifies both the number of bits written for `mark` and how these bits will be interpreted in an equivalent `fread` statement. `'float'` means single-precision numeric (usually 32 bits—4 bytes—although this value is hardware dependent). The default for this argument is `'uchar'`—unsigned characters (one byte).

- The statement

```
fclose(fid);
```

closes the file (returning `0` if the operation succeeds). Although MATLAB automatically closes all files when you exit from MATLAB it is good practice to close files explicitly with `fclose` when you have finished using them. Note that you can close all files with `fclose('all')`.

Reading binary data

The next program (`reader.m`) reads the file written with `writer.m` above and displays each record:

```
namelen = 10;                    % 10 bytes for name
fid = fopen('marks.bin', 'r');

while ~feof(fid)
    str = fread(fid,namelen);
    name = char(str');
    mark = fread(fid, 1, 'float');
    fprintf('%s %4.0f\n', name, mark)
end

fclose(fid);
```

Note:

- The file is opened for read only (`'r'`).
- The function `feof(fid)` returns `1` if the end of the specified file has been reached, and `0` otherwise.
- The first `fread` statement reads the next `namelen` (10) bytes from the file into the variable `str`. When the name was written by `fwrite` the ASCII codes of the characters were actually written to the file. Therefore the `char` function is needed to convert the codes back to characters. Furthermore the bytes being read are interpreted as entries in a column matrix; `str` must be transposed if you want to display the name is the usual horizontal format.
- The second `fread` statement specifies that one value (the number of values is given by the second argument) is to be read in `float` precision (four bytes). You could, for example, read an entire array of 78 `float` numbers with

```
a = fread(fid, 78, 'float');
```

Changing binary data

To demonstrate how to change records in a file we assume for simplicity that you will only want to change a student's mark, and not the name. The program below (`changer.m`) asks which record to change, displays the current name and mark in that record, asks for the corrected mark, and overwrites the original mark in that record.

```
namelen = 10;                    % 10 bytes for name
reclen = namelen + 4;
fid = fopen('marks.bin', 'r+'); % open for read and write

rec = input('Which record do you want to change?');
fpos = (rec-1)*reclen;      % file position indicator
fseek(fid, fpos, 'bof');    % move file position indicator
str = fread(fid,namelen);   % read the name
name = char(str');
mark = fread(fid, 1, 'float');  % read the mark
fprintf('%s %4.0f\n', name, mark)

mark = input('Enter corrected mark: '); % new mark
fseek(fid, -4, 'cof');       % go back 4 bytes to start of
                               mark
```

```
fwrite(fid, mark, 'float'); % overwrite mark
fprintf('Mark updated');

fclose(fid);
```

Note:

- The file is opened for reading *and* writing (`'r+'`).
- When a file is opened with `fopen` MATLAB maintains a *file position indicator*. The position in the file where MATLAB will begin the next operation on the file (reading or writing) is *one byte beyond* the file position indicator.

  `fpos` calculates the value of the file position indicator in order to commence reading the record number `rec`.
- The `fseek` function moves the file position indicator. Its second argument specifies where in the file to move the file position indicator, relative to an origin given by the third argument. Possible origins are `'bof'` (beginning of file), `'cof'` (current position in file) or `'eof'` (end of file).

  In this example, the records are 14 bytes long (10 for the name, four for the mark). If we wanted to update the second record, say, we would use

  ```
  fseek(fid, 14, 'bof');
  ```

  which moves the file position indicator to byte 14 from the beginning of the file, ready to start accessing at byte 15, which is the beginning of the second record. The function `fseek` returns `0` (successful) or `-1` (unsuccessful). Incidentally, if you get lost in the file you can always use `ftell` to find out where you are!
- The `fread` statement, which reads the mark to be changed, automatically advances the file position indicator by four bytes (the number of bytes required `float` precision). In order to overwrite the mark, we therefore have to move the file position indicator *back* four bytes from its current position. The statement

  ```
  fseek(fid, -4, 'cof');
  ```

  achieves this.
- The `fwrite` statement then overwrites the mark.

Note that the programs above don't have any error trapping devices, for example, preventing you from reading from a non-existent file, or preventing you from overwriting a record that isn't there. It is left to you to fill in these sort of details.

4.2.7 Other import/export functions

Other import/export functions, with differing degrees of flexibility and ease of use, include `csvread`, `csvwrite`, `dlmread`, `dlmwrite`, `fgets`, `fprintf` (which has an optional argument to specify a file), `fscanf`, `textread`, `xlsread`. You know where to look for the details!

Finally, recall that the `diary` command can also be used to export small arrays as text data, although you will need to edit out extraneous text.

Summary

- MATLAB functions may be used to perform a variety of mathematical, trigonometric and other operations.
- Data can be saved to disk files in text (ASCII) format or in binary format.
- `load` and `save` can be used to import/export both text and binary data (the latter in the form of MAT-files).
- The Import Wizard provides an easy way of importing both text and binary data.
- MATLAB's low-level I/O functions such as `fread` and `fwrite` provide random access to binary files.

EXERCISES

4.1 Write some MATLAB statements which will:

(a) find the length C of the hypotenuse of a right-angle triangle in terms of the lengths A and B of the other two sides;

(b) find the length C of a side of a triangle given the lengths A and B of the other two sides and the size in degrees of the included angle θ, using the cosine rule:

$$C^2 = A^2 + B^2 - 2AB\cos(\theta).$$

4.2 Translate the following formulae into MATLAB expressions:

(a) $\ln(x + x^2 + a^2)$

(b) $[e^{3t} + t^2\sin(4t)]\cos^2(3t)$

(c) $4\tan^{-1}(1)$ (inverse tangent)

(d) $\sec^2(x) + \cot(y)$

(e) $\cot^{-1}(|x/a|)$ (use MATLAB's inverse cotangent)

4.3 There are 39.37 inches in a meter, 12 inches in a foot, and three feet in a yard. Write a script to input a length in meters (which may have a decimal part) and convert it to yards, feet and inches. (Check: 3.51 meters converts to 3 yds 2 ft 6.19 in.)

4.4 A sphere of mass m_1 impinges obliquely on a stationary sphere of mass m_2, the direction of the blow making an angle α with the line of motion of the impinging sphere. If the coefficient of restitution is e it can be shown that the impinging sphere is deflected through an angle β such that

$$\tan(\beta) = \frac{m_2(1+e)\tan(\alpha)}{m_1 - em_2 + (m_1+m_2)\tan^2(\alpha)}$$

Write a script to input values of m_1, m_2, e, and α (in degrees) and to compute the angle β in degrees.

4.5 Section 2.6 has a program for computing the members of the sequence $x_n = a^n/n!$. The program displays every member x_n computed. Adjust it to display only every 10th value of x_n.

Hint: The expression `rem(n, 10)` will be zero only when `n` is an exact multiple of 10. Use this in an `if` statement to display every tenth value of x_n.

4.6 To convert the variable `mins` minutes into hours and minutes you would use `fix(mins/60)` to find the whole number of hours, and `rem(mins, 60)` to find the number of minutes left over. Write a script which inputs a number of minutes and converts it to hours and minutes.

Now write a script to convert seconds into hours, minutes and seconds. Try out your script on 10 000 seconds, which should convert to 2 hours 46 minutes and 40 seconds.

4.7 Design an algorithm (i.e. write the structure plan) for a machine which must give the correct amount of change from a $100 note for any purchase costing less than $100. The plan must specify the number and type of all notes and coins in the change, and should in all cases give as few notes and coins as possible. (If you are not familiar with dollars and cents, use your own monetary system.)

4.8 A uniform beam is freely hinged at its ends $x = 0$ and $x = L$, so that the ends are at the same level. It carries a uniformly distributed load of W per unit length, and there is a tension T along the x-axis. The deflection y of the beam a distance x from one end is given by

$$y = \frac{WEI}{T^2}\left[\frac{\cosh[a(L/2-x)]}{\cosh(aL/2)} - 1\right] + \frac{Wx(L-x)}{2T},$$

where $a^2 = T/EI$, E being the Young's modulus of the beam, and I is the moment of inertia of a cross-section of the beam. The beam is 10 m long, the tension is 1000 N, the load 100 N/m, and EI is 10^4.

Write a script to compute and plot a graph of the deflection y against x (MATLAB has a cosh function).

To make the graph look realistic you will have to override MATLAB's automatic axis scaling with the statement

```
axis([xmin xmax ymin ymax])
```

after the `plot` statement, where `xmin` etc. have appropriate values.

5

Logical vectors

The objectives of this chapter are to enable you to:

- **Understand logical operators more fully.**

And to introduce you to:

- **Logical vectors and how to use them effectively in a number of applications.**
- **Logical functions.**

This chapter introduces a most powerful and elegant feature of MATLAB, viz., the *logical vectors*. The topic is so useful and, hence, important that it deserves a chapter of its own.

Try these exercises on the command line:

1. Enter the following statements:

```
r = 1;
r <= 0.5      % no semicolon
```

   If you correctly left out the semicolon after the second statement you will have noticed that it returned the value 0.

2. Now enter the expression `r >= 0.5` (again, no semicolon). It should return the value 1. Note that we already saw in Chapter 2 that a logical expression in MATLAB involving only scalars returns a value of 0 if it is FALSE, and 1 if it is TRUE.

3. Enter the following statements:

```
r = 1:5;
r <= 3
```

Now the logical expression `r <= 3` (where `r` is a vector) returns a *vector*:

```
1     1     1     0     0
```

Can you see how to interpret this result? For each element of `r` for which `r <= 3` is true, 1 is returned; otherwise 0 is returned. Now enter `r == 4`. Can you see why `0  0  0  1  0` is returned?

When a vector is involved in a logical expression, the comparison is carried out *element by element* (as in an arithmetic operation). If the comparison is true for a particular element of the vector, the resulting vector, which is called a *logical vector*, has a 1 in the corresponding position; otherwise it has a 0. The same applies to logical expressions involving matrices. (Logical vectors were called 0-1 vectors in Version 4. Version 4 code involving 0-1 vectors will not run under Version 6 in certain circumstances—see below in Section 5.3.)

You can also compare vectors with vectors in logical expressions. Enter the following statements:

```
a = 1:5;
b = [0 2 3 5 6];
a == b      % no semicolon!
```

The logical expression `a == b` returns the logical vector

```
0     1     1     0     0
```

because it is evaluated element by element, i.e. `a(1)` is compared with `b(1)`, `a(2)` with `b(2)`, etc.

5.1 Examples

5.1.1 Discontinuous graphs

One very useful application of logical vectors is in plotting discontinuities. The following script plots the graph, shown in Figure 5.1, defined by

$$y(x) = \begin{cases} \sin(x) & (\sin(x) > 0) \\ 0 & (\sin(x) \leq 0) \end{cases}$$

over the range 0 to 3π:

```
x = 0 : pi/20 : 3 * pi;
y = sin(x);
```

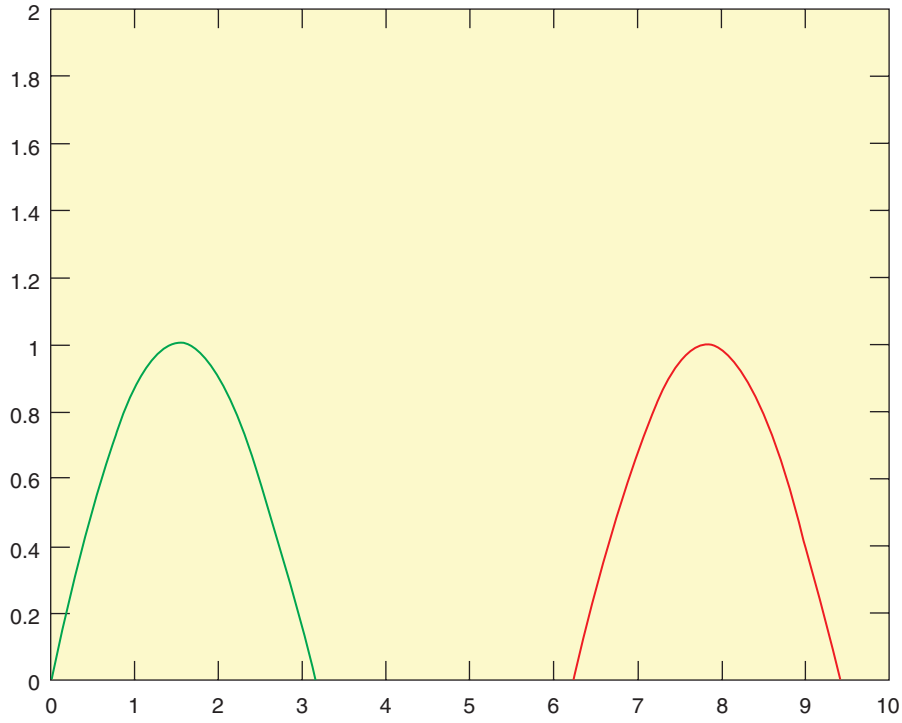


Figure 5.1 A discontinuous graph using logical vectors

```
y = y .* (y > 0); % set negative values of sin(x) to zero
plot(x, y)
```

The expression `y > 0` returns a logical vector with 1's where sin(*x*) is positive, and 0's otherwise. Element-by-element multiplication by `y` with `.*` then picks out the positive elements of `y`.

5.1.2 Avoiding division by zero

Suppose you want to plot the graph of $\sin(x)/x$ over the range -4π to 4π. The most convenient way to set up a vector of the *x* coordinates is

```
x = -4*pi : pi / 20 : 4*pi;
```

But then when you try

```
y = sin(x) ./ x;
```

you get the `Divide by zero` warning, because one of the elements of `x` is exactly zero. A neat way around this problem is to use a logical vector to replace

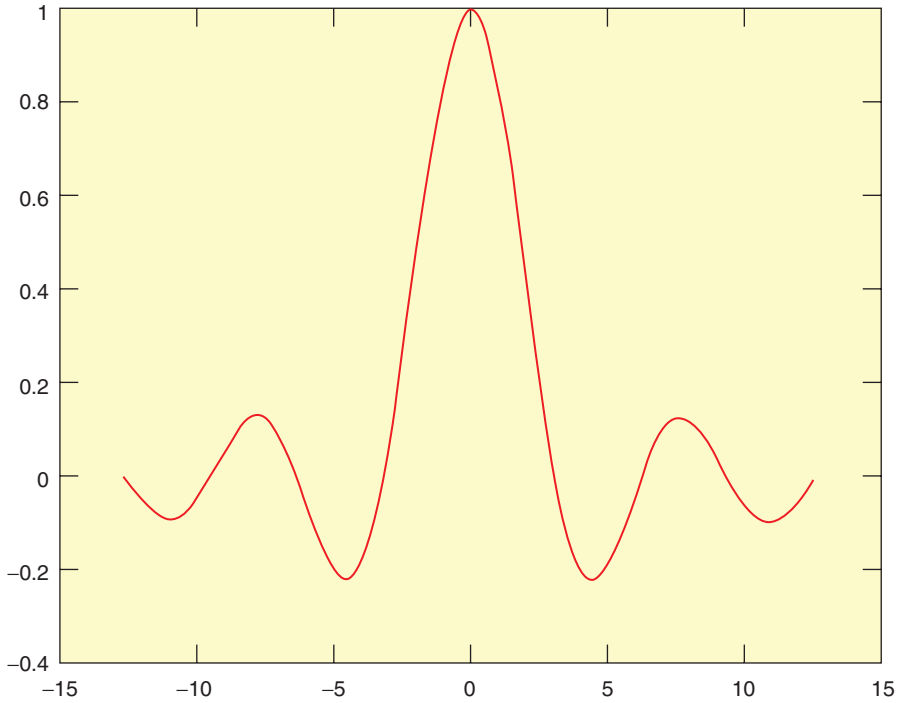


Figure 5.2 sin(*x*)/*x*

the zero with `eps`. This MATLAB function returns the difference between 1.0 and the next largest number which can be represented in MATLAB, i.e. approximately `2.2e-16`. Here is how to do it:

```
x = x + (x == 0)*eps;
```

The expression `x == 0` returns a logical vector with a single 1 for the element of `x` which is zero, and so `eps` is added only to that element. The following script plots the graph correctly—without a missing segment at $x = 0$ (see Figure 5.2).

```
x = -4*pi : pi/20 : 4*pi;
x = x + (x == 0)*eps;       % adjust x = 0 to x = eps
y = sin(x) ./ x;
plot(x, y)
```

When `x` has the value `eps`, the value of `sin(eps)/eps` has the correct limiting value of 1 (check it), instead of `NaN` (Not-a-Number) resulting from a division by zero.

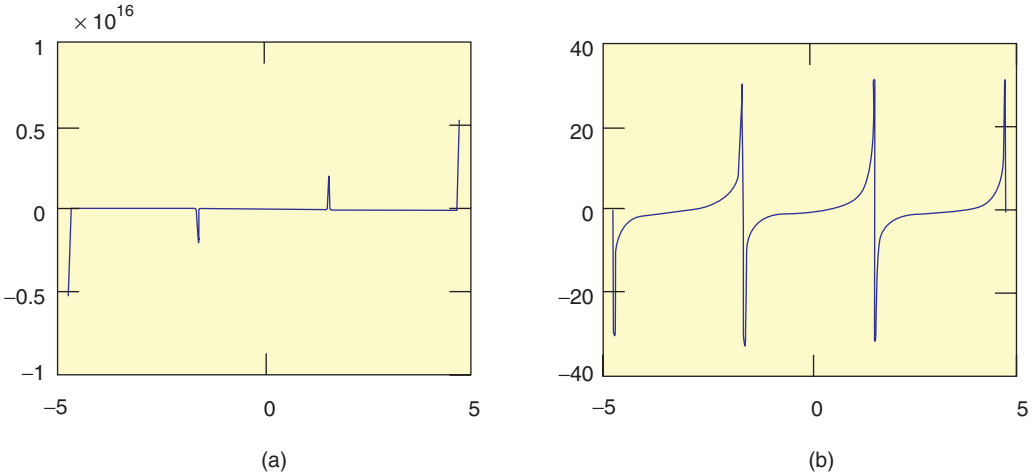


Figure 5.3 Variations on tan(*x*)

5.1.3 Avoiding infinity

The following script attempts to plot $\tan(x)$ over the range $-3\pi/2$ to $3\pi/2$. If you are not too hot at trig graphs, perhaps you should sketch the graph roughly with pen and paper before you run the script!

```
x = -3/2*pi : pi/100 : 3/2*pi;
y = tan(x);
plot(x, y)
```

The MATLAB plot—Figure 5.3(a)—should look nothing like your sketch. The problem is that $\tan(x)$ approaches $\pm\infty$ at odd multiples of $\pi/2$. The scale on the MATLAB plot is therefore very large (about 10^{15}), making it impossible to see the structure of the graph anywhere else.

If you add the statement

```
y = y .* (abs(y) < 1e10);    % remove the big ones
```

just before the `plot` statement you'll get a much nicer graph, as shown in Figure 5.3(b). The expression `abs(y) < 1e10` returns a logical vector which is zero only at the asymptotes. The graph thus goes through zero at these points, which incidentally draws nearly vertical asymptotes for you. These 'asymptotes' become more vertical as the increment in `x` becomes smaller.

5.1.4 Counting random numbers

The function `rand` returns a (pseudo-)random number in the interval [0, 1); `rand(1, n)` returns a row vector of `n` such numbers.

Try the following exercises on the command line:

1. Set up a vector `r` with seven random elements (leave out the semicolon so that you can see its elements):

   ```
   r = rand(1,7)        % no semicolon
   ```

   Check that the logical expression `r < 0.5` gives the correct logical vector.

2. Using the function `sum` on the logical expression `r < 0.5` will effectively count how many elements of `r` are less than 0.5. Try it, and check your answer against the values displayed for `r`:

   ```
   sum( r < 0.5 )
   ```

3. Now use a similar statement to count how many elements of `r` are greater than or equal to 0.5 (the two answers should add up to 7, shouldn't they?).

4. Since `rand` generates *uniformly distributed* random numbers, you would expect the number of random numbers which are less than 0.5 to get closer and closer to half the total number as more and more are generated.

   Generate a vector of a few thousand random numbers (suppress display with a semicolon this time) and use a logical vector to count how many are less than 0.5.

   Repeat a few times with a new set of random numbers each time. Because the numbers are random, you should never get quite the same answer each time.

Doing this problem without logical vectors is a little more involved. Here is the program:

```
tic                  % start
a = 0;               % number >= 0.5
b = 0;               % number < 0.5

for n = 1:5000
  r = rand;          % generate one number per loop
  if r >= 0.5
    a = a + 1;
```

```
  else
    b = b + 1;
  end;
end;
t = toc;                % finish
disp( ['less than 0.5: ' num2str(a)] )
disp( ['time: ' num2str(t)] )
```

It also takes a lot longer. Compare times for the two methods on your computer.

5.1.5 Rolling dice

When a fair dice is rolled, the number uppermost is equally likely to be any integer from 1 to 6. So if `rand` is a random number in the range [0, 1), `6 * rand` will be in the range [0, 6), and `6 * rand + 1` will be in the range [1, 7), i.e. between 1 and 6.9999. Discarding the decimal part of this expression with `floor` gives an integer in the required range. Try the following exercises:

1. Generate a vector `d` of 20 random integers in the range 1 to 6:

   ```
   d = floor(6 * rand(1, 20)) + 1
   ```

2. Count the number of 'sixes' thrown by summing the elements of the logical vector `d == 6`.
3. Verify your result by displaying `d`.
4. Estimate the probability of throwing a six by dividing the number of sixes thrown by 20. Using random numbers like this to mimic a real situation based on chance is called *simulation*.
5. Repeat with more random numbers in the vector `d`. The more you have, the closer the proportion of sixes gets to the theoretical expected value of 0.1667, i.e. 1/6.
6. Can you see why it would be incorrect to use `round` instead of `floor`? The problem is that `round` rounds in both directions, whereas `floor` rounds everything down.

5.2 Logical operators

We saw briefly in Chapter 2 that logical expressions can be constructed not only from the six relational operators, but also from the three *logical operators*

Table 5.1 Logical operators

Operator	Meaning
~	NOT
&	AND
|	OR

Table 5.2 Truth table (T = true; F = false)

lex1	lex2	~ lex1	lex1 & lex2	lex1 & lex2	xor(lex1, lex2)
F	F	T	F	F	F
F	T	T	F	T	T
T	F	F	F	T	T
T	T	F	T	T	F

shown in Table 5.1. Table 5.2 shows the effects of these operators on the general logical expressions *lex1* and *lex2*.

The OR operator (|) is technically an *inclusive* OR, because it is true when either or both of its operands are true. MATLAB also has an *exclusive* OR function `xor(a, b)` which is 1 (true) only when either *but not both of* `a` and `b` is 1 (Table 5.2).

MATLAB also has a number of functions that perform bitwise logical operations. See Help on `ops`.

The precedence levels of the logical operators, among others, are shown in Table 5.3. As usual, precedences may be overridden with brackets, e.g.

```
~ 0 & 0
```

returns 0 (false), whereas

```
~ (0 & 0)
```

returns 1 (true). Some more examples:

```
(b * b == 4 * a * c) & (a ~= 0)
(final >= 60) & (final < 70)
```

Table 5.3 Operator precedence (see **Help** on `operator precedence`)

Precedence	Operators
1.	`( )`
2.	`^ .^ ' .'` (pure transpose)
3.	+ (unary plus) - (unary minus) ~ (NOT)
4.	`* / \ .* ./ .\`
5.	+ (addition) - (subtraction)
6.	`:`
7.	`> < >= <= == ~=`
8.	& (AND)
9.	| (OR)

```
(a ~= 0) | (b ~= 0) | (c != 0)
~((a == 0) & (b == 0) & (c == 0))
```

It is never wrong to use brackets to make the logic clearer, even if they are syntactically unnecessary. Incidentally, the last two expressions above are logically equivalent, and are false only when `a=b=c=0`. It makes you think, doesn't it?

5.2.1 Operator precedence

You may accidentally enter an expression like

```
2 > 1 & 0
```

(try it) and be surprised because MATLAB (a) accepts it, and (b) returns a value of 0 (false). It is surprising because:

(a) `2 > 1 & 0` doesn't appear to make sense. If you have got this far you deserve to be let into a secret. MATLAB is based on the notorious language C, which allows you to mix different types of operators in this way (Pascal, for example, would never allow such flexibility!).

(b) We instinctively feel that `&` should have the higher precedence. `1 & 0` evaluates to 0, so `2 > 0` should evaluate to 1 instead of 0. The explanation is due partly to the resolution of surprise (a). MATLAB groups its operators in a rather curious and non-intuitive way. The complete operator precedence is given in Table 5.3 (reproduced for ease of reference in Appendix B). (Recall that the transpose operator `'` performs a *complex conjugate* transpose on complex data; the dot-transpose operator `.'` performs a 'pure' transpose without taking the complex conjugate.) Brackets always have the highest precedence.

5.2.2 Danger

I have seen quite a few students incorrectly convert the mathematical inequality $0 < r < 1$, say, into the MATLAB expression

```
0 < r < 1
```

Once again, the first time I saw this I was surprised that MATLAB did not report an error. Again, the answer is that MATLAB doesn't really mind how you mix up operators in an expression. It simply churns through the expression according to its rules (which may not be what you expect).

Suppose *r* has the value 0.5. Mathematically, the inequality is true for this value of *r* since it lies in the required range. However, the expression `0 < r < 1` is evaluated as 0. This is because the left-hand operation (`0 < 0.5`) is first evaluated to 1 (true), followed by `1 < 1` which is false.

Inequalities like this should rather be coded as

```
(0 < r) & (r < 1)
```

The brackets are not strictly necessary (see Table 5.3) but they certainly help to clarify the logic.

5.2.3 Logical operators and vectors

The logical operators can also operate on vectors (of the same size), returning logical vectors, e.g.

```
~(~[1 2 0 -4 0])
```

replaces all the non-zeros by 1's, and leaves the 0's untouched. Try it.

Note that brackets must be used to separate the two ~'s. Most operators may not appear directly next to each other (e.g. + ~); if necessary separate them with brackets.

The script in Section 5.1 that avoids division by zero has the critical statement

```
x = x + (x == 0)*eps;
```

This is equivalent to

```
x = x + (~x)*eps;
```

Try it, and make sure you understand how it works.

EXERCISE

Work out the results of the following expressions before checking them at the command line:

```
a = [-1 0 3];
b = [0 3 1];
~a
a & b
a | b
xor(a, b)
a > 0 & b > 0
a > 0 | b > 0
~ a > 0
a + (~ b)
a > ~ b
~ a > b
~ (a > b)
```

5.3 Subscripting with logical vectors

We saw briefly in Chapter 2 that elements of a vector may be referenced with subscripts, and that the subscripts themselves may be vectors, e.g.

```
a = [-2 0 1 5 9];
a([5 1 3])
```

returns

```
9    -2     1
```

i.e. the fifth, first and third elements of `a`. In general, if `x` and `v` are vectors, where `v` has *n* elements, then `x(v)` means

```
[x(v(1)), x(v(2)), ..., x(v(n))]
```

Now with `a` as defined above, see what

```
a(logical([0 1 0 1 0]))
```

returns. The function `logical(v)` returns a logical vector, with elements which are 1 or 0 according as the elements of `v` are non-zero or 0. (In Version 4 the `logical` function was not necessary here—you could use a '0-1' vector directly as a subscript.)

A summary of the rules for the use of a logical vector as a subscript are as follows:

- A logical vector `v` may be a subscript of another vector `x`.
- Only the elements of `x` corresponding to 1's in `v` are returned.
- `x` and `v` must be the same size.

Thus, the statement above returns

```
0     5
```

i.e. the second and fourth elements of `a`, corresponding to the 1's in `logical([0 1 0 1 0])`.

What will

```
a(logical([1 1 1 0 0]))
```

return? And what about `a(logical([0 0 0 0 0]))`?

Logical vector subscripts provide an elegant way of removing certain elements from a vector, e.g.

```
a = a(a > 0)
```

removes all the non-positive elements from `a`, because `a > 0` returns the logical vector `[0 0 1 1 1]`. We can verify incidentally that the expression `a > 0` is a logical vector, because the statement

```
islogical(a > 0)
```

returns 1. However, the *numeric* vector `[0 0 1 1 1]` is not a logical vector; the statement

```
islogical([0 0 1 1 1])
```

returns 0.

5.4 Logical functions

MATLAB has a number of useful *logical functions* that operate on scalars, vectors and matrices. Examples are given in the following list (where `x` is a vector unless otherwise stated). See Help on `logical functions`. (The functions are defined slightly differently for matrix arguments—see Chapter 6 or Help.)

`any(x)` returns the scalar 1 (true) if *any* element of `x` is non-zero (true).

`all(x)` returns the scalar 1 if *all* the elements of `x` are non-zero.

`exist('a')` returns 1 if `a` is a workspace variable. For other possible return values see `help`. Note that `a` must be enclosed in apostrophes.

`find(x)` returns a vector containing the subscripts of the *non-zero* (true) elements of `x`, so for example,

```
a = a( find(a) )
```

removes all the zero elements from `a`! Try it. Another use of `find` is in finding the subscripts of the largest (or smallest) elements in a vector, when there is more than one. Enter the following:

```
x = [8 1 -4 8 6];
find(x >= max(x))
```

This returns the vector `[1 4]`, which are the subscripts of the largest element (8). It works because the logical expression `x >= max(x)` returns a logical vector with 1's only at the positions of the largest elements.

`isempty(x)` returns 1 if `x` is an empty array and 0 otherwise. An empty array has a size of 0-by-0.

`isinf(x)` returns 1's for the elements of `x` which are `+Inf` or `-Inf`, and 0's otherwise.

`isnan(x)` returns 1's where the elements of `x` are `NaN` and 0's otherwise. This function may be used to remove `NaN`s from a set of data. This situation could arise while you are collecting statistics; missing or unavailable values can be temporarily represented by `NaN`s. However, if you do any calculations involving the `NaN`s, they propagate through intermediate calculations to the final result. To avoid this, the `NaN`s in a vector may be removed with a statement like

```
x(isnan(x)) = [ ]
```

MATLAB has a number of other logical functions starting with the characters `is`. See `is*` in the Help index for the complete list.

5.4.1 Using `any` and `all`

Because `any` and `all` with vector arguments return *scalars*, they are particularly useful in `if` statements. For example,

```
if all(a >= 1)
    do something
end
```

means 'if all the elements of the vector `a` are greater than or equal to 1, then *do something*'.

Recall from Chapter 2 that a vector condition in an `if` statement is true only if *all* its elements are non-zero. So if you want to execute *statement* below when two vectors `a` and `b` are equal (i.e. the same) you could say

```
if a == b
    statement
end
```

since `if` considers the logical vector returned by `a == b` true only if every element is a 1.

If, on the other hand, you want to execute *statement* specifically when the vectors `a` and `b` are *not* equal, the temptation is to say

```
if a ~= b      % wrong wrong wrong!!!
   statement
end
```

However this will *not* work, since *statement* will only execute if *each* of the corresponding elements of `a` and `b` differ. This is where `any` comes in:

```
if any(a ~= b)      % right right right!!!
   statement
end
```

This does what is required since `any(a ~= b)` returns the scalar 1 if *any* element of `a` differs from the corresponding element of `b`.

5.5 Logical vectors instead of `elseif` ladders

Those of us who grew up on more conventional programming languages in the last century may find it difficult to think in terms of using logical vectors when solving general problems. It's a nice challenge whenever writing a program to ask yourself whether you can possibly use logical vectors. They are almost always faster than other methods, although often not as clear to read later. You must decide when it's important for you to use logical vectors. But it's a very good programming exercise to force yourself to use them whenever possible! The following example illustrates these points by first solving a problem conventionally, and then with logical vectors.

It has been said that there are two unpleasant and unavoidable facts of life: death and income tax. A very simplified version of how income tax is calculated could be based on the following table:

Taxable income	**Tax payable**
$10 000 or less	10 percent of taxable income
between $10 000 and $20 000	$1000 + 20 percent of amount by which taxable income exceeds $10 000
more than $20 000	$3000 + 50 percent of amount by which taxable income exceeds $20 000

The tax payable on a taxable income of $30 000, for example, is

$$\$3000 + 50 \text{ percent of } (\$30\,000 - \$20\,000), \text{ i.e. } \$8000.$$

We would like to calculate the income tax on the following taxable incomes (in dollars): 5000, 10 000, 15 000, 30 000 and 50 000.

The conventional way to program this problem is to set up a vector with the taxable incomes as elements and to use a `loop` with an `elseif` ladder to process each element, as follows:

```
% Income tax the old-fashioned way

inc = [5000 10000 15000 30000 50000];

for ti = inc

  if ti < 10000
    tax = 0.1 * ti;
  elseif ti < 20000
    tax = 1000 + 0.2 * (ti - 10000);
  else
    tax = 3000 + 0.5 * (ti - 20000);
  end;

  disp( [ti tax] )
end;
```

Here is the output, suitably edited (note that the amount of tax paid changes continuously between tax 'brackets'—each category of tax is called a bracket):

```
Taxable income    Income tax

       5000.00        500.00
      10000.00       1000.00
      15000.00       2000.00
      30000.00       8000.00
      50000.00      18000.00
```

Now here is the logical way:

```
% Income tax the logical way

inc = [5000 10000 15000 30000 50000];

tax = 0.1 * inc .* (inc <= 10000);
tax = tax + (inc > 10000 & inc <= 20000) ...
```

```
                              .* (0.2 * (inc-10000) + 1000);
tax = tax + (inc > 20000) .* (0.5 * (inc-20000) + 3000);

disp( [inc' tax'] );
```

To understand how it works, it may help to enter the statements on the command line. Start by entering the vector `inc` as given. Now enter

```
inc <= 10000
```

which should give the logical vector `[1  1  0  0  0]`. Next enter

```
inc .* (inc <= 10000)
```

which should give the vector `[5000  10000  0  0  0]`. This has successfully selected only the incomes in the first bracket. The tax for these incomes is then calculated with

```
tax = 0.1 * inc .* (inc <= 10000)
```

which returns `[500  1000  0  0  0]`.

Now for the second tax bracket. Enter the expression

```
inc > 10000 & inc <= 20000
```

which returns the logical vector `[0  0  1  0  0]`, since there is only one income in this bracket. Now enter

```
0.2 * (inc-10000) + 1000
```

This returns `[0  1000  2000  5000  9000]`. Only the third entry is correct. Multiplying this vector by the logical vector just obtained blots out the other entries, giving `[0  0  2000  0  0]`. The result can be safely added to the vector `tax` since it will not affect the first two entries already there.

I am sure you will agree that the logical vector solution is more interesting than the conventional one!

Summary

- When a relational and/or logical operator operates on a vector expression, the operation is carried out element by element. The result is a logical vector consisting of 0's (FALSE) and 1's (TRUE).
- A vector may be subscripted with a logical vector of the same size. Only the elements corresponding to the 1's in the logical vector are returned.
- When one of the logical operators (`~ & |`) operates on an expression any non-zero value in an operand is regarded as TRUE; zero is regarded as FALSE. A logical vector is returned.
- Arithmetic, relational and logical operators may appear in the same expression. Great care must be taken in observing the correct operator precedence in such situations.
- Vectors in a logical expression must all be the same size.
- If a logical expression is a vector or a matrix, it is considered true in an `if` statement only if all its elements are non-zero.
- Brackets are needed to separate certain juxtaposed (adjacent) operators.
- The logical functions `any` and `all` return scalars when taking vector arguments, and are consequently useful in `if` statements.
- Logical vectors may often be used instead of the more conventional `elseif` ladder. This provides faster more elegant code, but requires more ingenuity and the code may be less clear to read later on.

EXERCISES

5.1 Determine the values of the following expressions yourself before checking your answers using MATLAB. You may need to consult Table 5.3.

(a) `1 & -1`
(b) `13 & ~(-6)`
(c) `0 < -2|0`
(d) `~[1 0 2] * 3`
(e) `0 <= 0.2 <= 0.4`
(f) `5 > 4 > 3`
(g) `2 > 3 & 1`

5.2 Given that `a = [1 0 2]` and `b = [0 2 2]` determine the values of the following expressions. Check your answers with MATLAB.

(a) `a ~= b`
(b) `a < b`
(c) `a < b < a`
(d) `a < b < b`
(e) `a | (~a)`
(f) `b & (~b)`
(g) `a(~(~b))`
(h) `a = b == a` (determine final value of `a`)

5.3 Write some MATLAB statements on the command line which use logical vectors to count how many elements of a vector `x` are negative, zero or positive. Check that they work, e.g. with the vector

```
[-4 0 5 -3 0 3 7 -1 6]
```

5.4 The Receiver of Revenue (Internal Revenue Service) decides to change the tax table used in Section 5.5 slightly by introducing an extra tax bracket and changing the tax-rate in the third bracket, as follows:

Taxable income	**Tax payable**
$10 000 or less	10 percent of taxable income
between $10 000 and $20 000	$1000 + 20 percent of amount by which taxable income exceeds $10 000
between $20 000 and $40 000	$3000 + 30 percent of amount by which taxable income exceeds $20 000
more than $40 000	$9000 + 50 percent of amount by which taxable income exceeds $40 000

Amend the logical vector script to handle this table, and test it on the following list of incomes (dollars): 5000, 10 000, 15 000, 22 000, 30 000, 38 000 and 50 000.

5.5 A certain company offers seven annual salary levels (dollars): 12 000, 15 000, 18 000, 24 000, 35 000, 50 000 and 70 000. The number of employees paid at each level are, respectively: 3000, 2500, 1500, 1000, 400, 100 and 25. Write some statements at the command line to find the following:

(a) The average salary *level*. Use `mean`. (Answer: 32 000)

(b) The number of employees above and below this average salary level. Use logical vectors to find which salary levels are above and below the average level. Multiply these logical vectors element by element with the employee vector, and `sum` the result. (Answer: 525 above, 8000 below)

(c) The *average salary earned* by an individual in the company (i.e. the total annual salary bill divided by the total number of employees). (Answer: 17 038.12)

5.6 Write some statements on the command line to remove the largest element(s) from a vector. Try it out on `x = [1 2 5 0 5]`. The idea is to end up with `[1 2 0]` in `x`. Use `find` and the empty vector `[ ]`.

5.7 The electricity accounts of residents in a very small rural community are calculated as follows:

- if 500 units or less are used the cost is 2 cents per unit;
- if more than 500, but not more than 1000 units are used, the cost is $10 for the first 500 units, and then 5 cents for every unit in excess of 500;
- if more than 1000 units are used, the cost is $35 for the first 1000 units plus 10 cents for every unit in excess of 1000;
- in addition, a basic service fee of $5 is charged, no matter how much electricity is used.

The five residents use the following amounts (units) of electricity in a certain month: 200, 500, 700, 1000 and 1500. Write a program which uses logical vectors to calculate how much they must pay. Display the results in two columns: one for the electricity used in each case, and one for amount owed. (Answers: $9, $15, $25, $40, $90)

6

Matrices of numbers & arrays of strings

The objectives of this chapter are to:

- **Introduce you to ways of creating and manipulating matrices.**
- **Introduce you to matrix operations.**
- **Introduce you to character strings and facilities for handling them.**

As we have already seen, the name MATLAB stands for MATrix LABoratory, because the MATLAB system is designed specially to work with data arranged in the form of *matrices* (2-D arrays) . The word 'matrix' has two distinct meanings in this chapter:

1. An arrangement of data in rows and columns, e.g. a table.
2. A mathematical object, for which particular mathematical operations are defined, e.g. 'matrix' multiplication.

We begin this chapter (in Sections 6.1, 6.2, & 6.3) by looking at matrices in the first sense, summarizing and extending what we learned about them in Chapter 2. We then go on to look briefly at the mathematical operations on matrices. In a later chapter we will see how these operations can be applied in a number of widely differing areas, such as systems of linear equations, population dynamics, and Markov processes.

Next (in Sections 6.4, 6.5, & 6.6), we discuss arrays of *string* variables. A *string* in MATLAB is a collection of characters enclosed in apostrophes (single quotes). Strings are also referred to as *text*); this kind of data type is described in a little more detail at the end of this chapter.

6.1 Matrices

6.1.1 A concrete example

A ready-mix concrete company has three factories (S1, S2 and S3) which must supply three building sites (D1, D2 and D3). The costs, in some suitable currency, of transporting a load of concrete from any factory to any site are given by the following cost table:

	D1	*D2*	*D3*
S1	3	12	10
S2	17	18	35
S3	7	10	24

The factories can supply 4, 12 and 8 loads per day, respectively, and the sites require 10, 9 and 5 loads per day, respectively. The real problem is to find the cheapest way to satisfy the demands at the sites, but we are not considering that here.

Suppose the factory manager proposes the following transportation scheme (each entry represents the number of loads of concrete to be transported along that particular route):

	D1	*D2*	*D3*
S1	4	0	0
S2	6	6	0
S3	0	3	5

This sort of scheme is called a *solution* to the transportation problem. The cost table (and the solution) can then be represented by tables **C** and **X**, say, where c_{ij} is the entry in row i and column j of the cost table, with a similar convention for **X**.

To compute the cost of the above solution, each entry in the solution table must be multiplied by the corresponding entry in the cost table. (This operation is not

to be confused with the mathematical operation of matrix multiplication, which is discussed later.) We therefore want to calculate

$$3 \times 4 + 12 \times 0 + \cdots + 24 \times 5.$$

To do this calculation in MATLAB, enter **C** and **X** as matrices from the command line, with a semi-colon at the end of each row:

```
   c = [3 12 10; 17 18 35; 7 10 24];
x = [4 0 0; 6 6 0; 0 3 5];
```

and then find the *array* product of `c` and `x`:

```
total = c .* x
```

which gives

```
 12     0     0
102   108     0
  0    30   120
```

The command

```
sum(total)
```

then returns a vector where each element is the sum of each column of `total`:

```
114   138   120
```

Summing this in turn, i.e. `sum(sum( total ))`, gives the final answer of 372.

6.1.2 Creating matrices

As we saw above, use a semicolon to indicate the end of a row when entering a matrix. Bigger matrices can be constructed from smaller ones, e.g. the statements

```
a = [1 2; 3 4];
x = [5 6];
a = [a; x]
```

result in

```
a =
    1     2
    3     4
    5     6
```

Instead of a semicolon, you can use a line-feed (**Enter**) to indicate the end of a row.

6.1.3 Subscripts

Individual elements of a matrix are referenced with two subscripts, the first for the row, and the second for the column, e.g. `a(3,2)` for `a` above returns 6.

Alternatively, and less commonly, a single subscript may be used. In this case you can think of the matrix as being 'unwound' column by column. So `a(5)` for the above example returns 4.

If you refer to a subscript which is out of range, e.g. `a(3,3)` for `a` above, you will get an error message. However, if you assign a value to an element with a subscript which is out of range the matrix is enlarged to accommodate the new element. For example, the assignment

```
a(3,3) = 7
```

will add a third column to `a` with 0's everywhere except at `a(3,3)`.

6.1.4 Transpose

The statements

```
a = [1:3; 4:6]
b = a'
```

result in

```
a =
     1     2     3
     4     5     6

b =
     1     4
     2     5
     3     6
```

The transpose operator `'` (apostrophe) turns rows into columns and vice versa.

6.1.5 The colon operator

➤ The colon operator is extremely powerful, and provides for very efficient ways of handling matrices, e.g. if `a` is the matrix

```
a =
     1     2     3
     4     5     6
     7     8     9
```

the statement

```
a(2:3,1:2)
```

results in

```
4     5
7     8
```

(returns second and third rows, first and second columns), the statement

```
a(3,:)
```

results in

```
7     8     9
```

(returns third row), and the statement

```
a(1:2,2:3) = ones(2)
```

results in

```
a =
    1     1     1
    4     1     1
    7     8     9
```

(replaces the 2-by-2 submatrix composed of the first and second row and the second and third column with a square matrix of 1's).

Essentially, what is happening in the above examples is that the colon operator is being used to create vector subscripts. However, a colon by itself in place of a subscript denotes *all* the elements of the corresponding row or column. So `a(3,:)` means all elements in the third row.

This feature may be used, for example, to construct tables. Suppose we want a table `trig` of the sines and cosines of the angles 0° to 180° in steps of 30°. The following statements achieve this:

```
x = [0:30:180]';
trig(:,1) = x;
trig(:,2) = sin(pi/180*x);
trig(:,3) = cos(pi/180*x);
```

- You can use vector subscripts to get more complicated effects, e.g.

```
a(:,[1 3]) = b(:,[4 2])
```

replaces the first and third columns of `a` by the fourth and second columns of `b` (`a` and `b` must have the same number of rows).

- The colon operator is ideal for the sort of row operations performed in Gauss reduction (a technique of numerical mathematics), for example. If `a` is the matrix

```
a =
     1    -1     2
     2     1    -1
     3     0     1
```

the statement

```
a(2,:) = a(2,:) - a(2,1)*a(1,:)
```

subtracts the first row multiplied by the first element in the second row from the second row, resulting in

```
a =
     1    -1     2
     0     3    -5
     3     0     1
```

(the idea being to get a zero immediately underneath `a(1,1)`).

- The keyword `end` refers to the *last* row or column of an array. For example, if `r` is a row vector, the statement

```
sum(r(3:end))
```

returns the sum of all the elements of `r` from the third one to the last one.

- The colon operator may also be used as a single subscript, in which case it behaves differently if it is on the right-hand or left-hand side of an assignment. On the right-hand side of an assignment, `a(:)` gives all the elements of `a` strung out *by columns* in one long column vector, e.g. if

```
a =
     1     2
     3     4
```

the statement

```
b = a(:)
```

results in

```
b =
     1
     3
     2
     4
```

However, on the left-hand side of an assignment, `a(:)` *reshapes* a matrix. `a` must already exist. Then `a(:)` denotes a matrix with the same dimensions (shape) as `a`, but with new contents taken from the right-hand side. In other words, the matrix on the right-hand side is reshaped into the shape of `a` on the left-hand side. Some examples may help. If

```
b =
     1     2     3
     4     5     6
```

and

```
a =
     0     0
     0     0
     0     0
```

the statement

```
a(:) = b
```

results in

```
a =
     1     5
     4     3
     2     6
```

(the contents of `b` are strung out into one long column and then fed into `a` by columns). As another example, the statement

```
a(:) = 1:6
```

(with `a` as above) results in

```
a =
     1     4
     2     5
     3     6
```

Reshaping can also be done with the `reshape` function. See `help`.

Whether the colon operator appears on the right- or left-hand side of an assignment, the matrices or submatrices on each side of the assignment have the same shape (except in the special case below).

- As a special case, a single colon subscript may be used to replace *all* the elements of a matrix with a scalar, e.g.

```
a(:) = -1
```

6.1.6 Duplicating rows and columns: tiling

Sometimes it is useful to generate a matrix where all the rows or columns are the same. This can be done with the `repmat` function as follows. If `a` is the row vector

```
a =
     1     2     3
```

the statement

```
repmat(a, [3 1])
```

results in

```
ans =
     1     2     3
     1     2     3
     1     2     3
```

In `help`'s inimitable phraseology, this statement produces a 3-by-1 'tiling' of copies of `a`. You can think of `a` as a 'strip' of three tiles stuck to self-adhesive backing. The above statement tiles a floor with three rows and one column of such a strip of tiles.

There is an alternative syntax for `repmat`:

```
repmat(a, 3, 1)
```

An interesting example of this process appears at the end of this section in the loan repayment problem.

6.1.7 Deleting rows and columns

Use the colon operator and the empty array to delete entire rows or columns. For example,

```
a(:,2) = [ ]
```

deletes the second column of `a`.

You can't delete a single element from a matrix while keeping it a matrix, so a statement like

```
a(1,2) = [ ]
```

results in an error. However, using the single subscript notation you can delete a sequence of elements from a matrix and reshape the remaining elements into a row vector. For example, if

```
a =
     1     2     3
     4     5     6
     7     8     9
```

the statement

```
a(2:2:6) = [ ]
```

results in

```
a =
     1     7     5     3     6     9
```

Get it? (First unwind `a` by columns, then remove elements 2, 4 and 6.)

You can use logical vectors to extract a selection of rows or columns from a matrix. For example, if `a` is the original 3-by-3 matrix defined above, the statement

```
a(:, logical([1 0 1]))
```

results in

```
ans =
       1     3
       4     6
       7     9
```

(first and third columns extracted). The same effect is achieved with

```
a(:, [1 3])
```

6.1.8 Elementary matrices

There is a group of functions to generate 'elementary' matrices which are used in a number of applications. See `help elmat`.

For example, the functions `zeros`, `ones` and `rand` generate matrices of 1's, 0's and random numbers, respectively. With a single argument *n*, they generate

$n \times n$ (square) matrices. With two arguments n and m they generate $n \times m$ matrices. (For very large matrices `repmat` is usually faster than `ones` and `zeros`.)

The function `eye(n)` generates an $n \times n$ *identity* matrix, i.e. a matrix with 1's on the main 'diagonal', and 0's everywhere else, e.g. the statement

```
eye(3)
```

results in

```
ans =
     1     0     0
     0     1     0
     0     0     1
```

(The original version of MATLAB could not use the more obvious name `I` for the identity since it did not distinguish between upper- and lowercase letters and `i` was the natural choice for the imaginary unit number.)

As an example, `eye` may be used to construct a *tridiagonal* matrix as follows. The statements

```
a = 2 * eye(5);
a(1:4, 2:5) = a(1:4, 2:5) - eye(4);
a(2:5, 1:4) = a(2:5, 1:4) - eye(4)
```

result in

```
a =
     2    -1     0     0     0
    -1     2    -1     0     0
     0    -1     2    -1     0
     0     0    -1     2    -1
     0     0     0    -1     2
```

Incidentally, if you work with large tridiagonal matrices, you should have a look at the sparse matrix facilities available in MATLAB via the help browser.

6.1.9 *Specialized matrices

The following are functions that could be used to generate arbitrary matrices to investigate matrix operations when you cannot think of one to generate yourself. They are special matrices that were discovered by mathematicians, the

motivation for their discovery and other details about them you are *not* expected to understand.

`pascal(n)` generates a Pascal matrix of order *n*. Technically, this is a symmetric positive definite matrix with entries made up from *Pascal's triangle*, e.g.

```
pascal(4)
```

results in

```
ans =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

`magic(n)` generates an $n \times n$ *magic square*.

MATLAB has a number of other functions which generate special matrices, such as `gallery`, `hadamard`, `hankel`, `hilb`, `toeplitz`, `vander`, etc. See `help elmat`.

6.1.10 Using MATLAB functions with matrices

When a MATLAB mathematical or trigonometric function has a matrix argument, the function operates on every element of the matrix, as you would expect. However, many of the other MATLAB functions operate on matrices *column by column*, e.g. if

```
a =
     1     0     1
     1     1     1
     0     0     1
```

the statement

```
all(a)
```

results in

```
ans =
     0     0     1
```

For each *column* of `a` where all the elements are true (non-zero) `all` returns 1, otherwise it returns 0. `all` therefore returns a logical vector when it takes

a matrix argument. To test if all the elements of `a` are true, use `all` twice. In this example, the statement

```
all(all(a))
```

returns 0 because some of the elements of `a` are 0. On the other hand the statement

```
any(a)
```

returns

```
ans =
     1     1     1
```

because each column of `a` has at least one non-zero element, and `any(any(a))` returns 1, since `a` itself has at least one non-zero element.

If you are not sure whether a particular function operates columnwise or element by element on matrices, you can always request `help`.

6.1.11 Manipulating matrices

Here are some functions for manipulating matrices. See `help` for details.

`diag` extracts or creates a diagonal.

`fliplr` flips from left to right.

`flipud` flips from top to bottom.

`rot90` rotates.

`tril` extracts the lower triangular part, e.g. the statement

```
tril(pascal(4))
```

results in

```
ans =
     1     0     0     0
     1     2     0     0
     1     3     6     0
     1     4    10    20
```

`triu` extracts the upper triangular part.

6.1.12 Array (element-by-element) operations on matrices

The array operations discussed in Chapter 2 all apply to matrices as well as vectors. For example, if `a` is a matrix, `a * 2` multiplies each element of `a` by 2, and if

```
a =
     1     2     3
     4     5     6
```

the statement

```
a .^ 2
```

results in

```
ans =
      1     4     9
     16    25    36
```

6.1.13 Matrices and `for`

If

```
a =
     1     2     3
     4     5     6
     7     8     9
```

the statements

```
for v = a
  disp(v')
end
```

result in

```
1     4     7
2     5     8
3     6     9
```

What happens in this most general form of the `for` statement is that the index `v` takes on the value of each *column* of the matrix expression `a` in turn. This

provides a neat way of processing all the columns of a matrix. You can do the same with the rows if you transpose `a`, e.g. the statements

```
for v = a'
  disp(v')
end
```

display the rows of `a` one at a time. Get it?

6.1.14 Visualization of matrices

Matrices can be visualized graphically in MATLAB. This subject is discussed briefly in Chapter 7, with illustrations.

6.1.15 Vectorizing nested `for`s: loan repayment tables

If a regular fixed payment P is made n times a year to repay a loan of amount A over a period of k years, where the nominal annual interest rate is r, P is given by

$$P = \frac{rA(1 + r/n)^{nk}}{n[(1 + r/n)^{nk} - 1]}.$$

We would like to generate a table of repayments for a loan of $1000 over 15, 20 or 25 years, at interest rates that vary from 10 to 20 percent per annum, in steps of 1 percent. Since P is directly proportional to A in the above formula, the repayments of a loan of any amount can be found by simple proportion from such a table.

The conventional way of handling this is with 'nested' `for`s. The `fprintf` statements are necessary in order to get the output for each interest rate on the same line (see Section 2.9):

```
A = 1000;                % amount borrowed
n = 12;                  % number of payments per year

for r = 0.1 : 0.01 : 0.2
  fprintf( '%4.0f%', 100 * r );
  for k = 15 : 5 : 25
    temp = (1 + r/n) ^ (n*k);
    P = r * A * temp / (n * (temp - 1));
    fprintf( '%10.2f', P );
  end;
  fprintf( ' \n' );        % new line
end;
```

Some sample output (with headings edited in):

```
rate %    15 yrs     20 yrs     25 yrs

  10      10.75       9.65       9.09
  11      11.37      10.32       9.80
  ...
  19      16.83      16.21      15.98
  20      17.56      16.99      16.78
```

However, we saw in Chapter 2 that `for` loops can often be vectorized, saving a lot of computing time (and also providing an interesting intellectual challenge!). The inner loop can easily be vectorized; the following code uses only one `for`:

```
...
for r = 0.1 : 0.01 : 0.2
  k = 15 : 5 : 25;
  temp = (1 + r/n) .^ (n*k);
  P = r * A * temp / n ./ (temp - 1);
  disp( [100 * r, P] );
end;
```

Note the use of the array operators.

The really tough challenge, however, is to vectorize the outer loop as well. We want a table with 11 rows and 3 columns. Start by assigning values to `A` and `n` (from the command line):

```
A = 1000;
n = 12;
```

Then generate a column vector for the interest rates:

```
r = [0.1:0.01:0.2]'
```

Now change this into a table with 3 columns each equal to `r`:

```
r = repmat(r, [1 3])
```

The matrix `r` should look like this:

```
0.10          0.10          0.10
0.11          0.11          0.11
...
0.19          0.19          0.19
0.20          0.20          0.20
```

Now do a similar thing for the repayment periods `k`. Generate the row vector

```
k = 15:5:25
```

and expand it into a table with 11 rows each equal to `k`:

```
k = repmat(k, [11 1])
```

This should give you

```
15    20    25
15    20    25
...
15    20    25
15    20    25
```

The formula for `P` is a little complicated, so let's do it in two steps:

```
temp = (1 + r/n) .^ (n * k);
P = r * A .* temp / n ./ (temp - 1)
```

Finally, you should get this for `P`:

```
10.75          9.65          9.09
11.37         10.32          9.80
...
16.83         16.21         15.98
17.56         16.99         16.78
```

This works, because of the way the tables `r` and `k` have been constructed, and because the MATLAB array operations are performed element by element. For example, when the calculation is made for `P` in row 2 and column 1, the array operations pick out row 2 of `r` (all 0.11) and column 1 of `k` (all 15), giving the correct value for `P` (11.37).

The nested `for` way might be easier to program, but this way is certainly more interesting (and quicker to execute).

6.1.16 Multidimensional arrays

MATLAB arrays can have more than two dimensions. For example, suppose you create the matrix

```
a = [1:2; 3:4]
```

You can add a third dimension to `a` with

```
a(:,:,2) = [5:6; 7:8]
```

MATLAB responds with

```
a(:,:,1) =
     1     2
     3     4
a(:,:,2) =
     5     6
     7     8
```

It helps to think of the 3-D array `a` as a series of 'pages', with a matrix on each page. The third dimension of `a` numbers the pages. This is analogous to a spreadsheet with multiple sheets: each sheet contains a table (matrix).

You can get into hyperpages with higher dimensions if you like!

See `help datatypes` for a list of special multidimensional array functions.

6.2 Matrix operations

We have seen that *array* operations are performed element by element on matrices. However, *matrix* operations, which are fundamental to MATLAB, are defined differently in certain cases, in the mathematical sense.

Matrix addition and subtraction are defined in the same way as the equivalent array operations, i.e. element by element. Matrix multiplication, however, is quite different.

6.2.1 Matrix multiplication

Matrix *multiplication* is probably the most important matrix operation. It is used widely in such areas as network theory, solution of linear systems of equations, transformation of coordinate systems, and population modeling, to name but a very few. The rules for multiplying matrices look a little weird if you've never seen them before, but will be justified by the applications that follow.

When two matrices **A** and **B** are multiplied together in this sense, their product is a third matrix **C**. The operation is written as

$$\mathbf{C} = \mathbf{AB},$$

and the general element c_{ij} of **C** is formed by taking the *scalar product* of the ith row of **A** with the jth column of **B**. (The scalar product of two *vectors* **x** and **y** is $x_1y_1 + x_2y_2 + \cdots$, where x_i and y_i are the components of the vectors.) It follows that **A** and **B** can only be successfully multiplied (in that order) if the number of columns in **A** is the same as the number of rows in **B**.

The general definition of matrix multiplication is as follows: If **A** is a $n \times m$ matrix and **B** is a $m \times p$ matrix, their product **C** will be a $n \times p$ matrix such that the general element c_{ij} of **C** is given by

$$c_{ij} = \sum_{k=1}^{m} a_{ik}\, b_{kj}.$$

Note that in general **AB** is not equal to **BA** (matrix multiplication is not *commutative*).

Example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 15 & 14 \end{bmatrix}$$

Since a vector is simply a one-dimensional matrix, the definition of matrix multiplication given above also applies when a vector is multiplied by an appropriate matrix, e.g.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 8 \\ 18 \end{bmatrix}.$$

The operator `*` is used for matrix multiplication, as you may have guessed. For example, if

```
a =
     1     2
     3     4
```

and

```
b =
     5     6
     0    -1
```

the statement

```
c = a * b
```

results in

```
c =
     5     4
    15    14
```

Note the important difference between the *array* operation `a .* b` (evaluate by hand and check with MATLAB) and the *matrix* operation `a * b`.

To multiply a matrix by a vector in that order, the vector must be a column vector. So if

```
b = [2 3]'
```

the statement

```
c = a * b
```

results in

```
c =
     8
    18
```

6.2.2 Matrix exponentiation

The matrix operation $\mathbf{A}^2$ means $\mathbf{A} \times \mathbf{A}$, where $\mathbf{A}$ must be a square matrix. The operator `^` is used for matrix exponentiation, e.g. if

```
a =
     1     2
     3     4
```

the statement

```
a ^ 2
```

results in

```
ans =
     7    10
    15    22
```

which is the same as `a * a` (try it).

Again, note the difference between the array operation `a .^ 2` (evaluate by hand and check with MATLAB) and the matrix operation `a ^ 2`.

6.3 Other matrix functions

Here are some of MATLAB's more advanced matrix functions.

`det` determinant.

`eig` eigenvalue decomposition.

`expm` matrix exponential, i.e. $e^{\mathbf{A}}$, where **A** is a matrix. The matrix exponential may be used to evaluate analytical solutions of linear ordinary differential equations with constant coefficients.

`inv` inverse.

`lu` LU factorization (into lower and upper triangular matrices).

`qr` orthogonal factorization.

`svd` singular value decomposition.

6.4 *Strings

6.4.1 Assignment

Strings may be assigned to variables by enclosing them in apostrophes:

```
s = 'Hi there';
```

If an apostrophe is to be included in the string it must be repeated:

```
s = 'o''clock';
```

6.4.2 Input

Strings may be entered in response to the `input` statement in two ways:

1. You can enclose the string in apostrophes when you enter it, or

2. you can use an additional argument `'s'` with `input`, in which case you must not use apostrophes when entering the string, e.g.

```
>> name = input( 'Enter your surname: ', 's' );
Enter your surname: O'Reilly
```

6.4.3 Strings are arrays

A MATLAB string is actually an array with each element representing one character in the string. For example, if

```
s = 'Napoleon'
```

`whos` reveals that `s` is 1-by-8. The statement

```
s(8:-1:1)
```

will therefore display the string '`Napoleon`' backwards.

If a string must contain an apostrophe, the apostrophe must be repeated when entering the string. However, only one apostrophe is actually stored in the string. For example, in

```
name = 'M''kombe'
```

`name(3)` is '`k`'.

You can exploit the vector nature of strings in a number of ways, e.g. to find and/or remove certain characters. The following code will remove all the blanks from the string `s`.

```
s = 'Twas brillig and the slithy toves';
nonblanks = s ~= ' ';
s(nonblanks)
```

`nonblanks` is a logical vector with 0's corresponding to the positions of the blanks in `s`. Using `nonblanks` as a subscript then removes the blanks (**Subscripting with logical vectors**, Chapter 5).

6.4.4 Concatenation of strings

Because strings are vectors, they may be *concatenated* (joined) with square brackets, e.g.

```
king = 'Henry';
king = [king, ' VIII']
```

6.4.5 ASCII codes, `double` and `char`

A MATLAB character is represented internally by a 16-bit numerical value. The character codes start at 1; the first 127 such codes are the *ASCII* codes. For example, the ASCII codes for the letters 'A' to 'Z' are the consecutive integers from 65 to 90, while the codes for 'a' to 'z' run from 97 to 122. (Appendix D gives all the ASCII codes.)

You can see the ASCII codes for a string with `double`, e.g.

```
double('Napoleon')
```

returns

```
78    97   112   111   108   101   111   110
```

Conversely, `char` returns a string from the ASCII codes, e.g.

```
char(65:70)
```

returns

```
ABCDEF
```

You can use `char` and `double` to generate rows of identical characters, e.g.

```
x = char(ones(4,20)*double('#'))
```

gives

```
####################
####################
####################
####################
```

Can you see why? The function `ones(4,20)` generates 4 rows and 20 columns of 1's. Multiplying by `double('#')` replaces all the 1's with the ASCII code for # (whatever that is), and `char` converts the ASCII codes back to text.

If a character variable is involved in an arithmetic expression, MATLAB uses the ASCII code of the character in the calculation, e.g. if

```
s = 'a'
```

the expression `s + 1` returns 98, since the ASCII code for 'a' is 97.

Array operations can be performed on strings. For example, if `s` is a string of letters, the expression `char(s+1)` returns each letter in the string advanced by one place in the alphabet.

The relationship between characters and their ASCII codes means you can do neat things like this:

```
alpha = double('a'):double('z')
```

Can you see what the result will be?

6.4.6 `fprintf` of strings

Strings may be displayed with `fprintf` using the `%s` format specifier. They can be right- or left-justified, and truncated, e.g. the statements

```
fprintf( '%8sVIII\n', 'Henry' )
fprintf( '%-8sVIII\n', 'Henry' )
fprintf( '%.3sVIII\n', 'Henry' )
```

result in (the boxes indicate spaces)

```
   HenryVIII
HenryVIII
HenVIII
```

To display single characters with `fprintf` use the `%c` format specifier.

6.4.7 Comparing strings

Try out the following statements:

```
s1 = 'ann';
s2 = 'ban';
s1 < s2
```

You should get the logical vector

```
1     0     0
```

because the logical expression `s1 < s2` compares the ASCII codes of the two strings element by element (the strings must have the same length). This is the basis of alphabetical sorting.

The function `strcmp(s1, s2)` compares the two strings `s1` and `s2`, returning 1 if they are identical, and 0 otherwise. (The strings do not have to be the same length.)

6.4.8 Other string functions

`blanks`
generates a string of blanks.

`deblank`
removes trailing blanks from a string.

`int2str`, `num2str`
convert their numeric arguments to strings. These functions are handy for labeling graphs with text which includes variable numeric values.

`ischar`
returns 1 if its argument is a string, and 0 otherwise.

`lower`, `upper`
convert strings to lowercase and uppercase, respectively.

`sprintf`
works like `fprintf`, except that it returns a string according to the format specification, e.g.

```
s = sprintf( 'Value of K: %g', K )
```

returns the string `Value of K: 0.015'` in `s` if `K` has the value 0.015.

See `help matlab\strfun` for a complete list of string-handling functions.

6.5 *Two-dimensional strings

You can create a two-dimensional character array in the same way as you would a matrix, except that each row must have the same length; if necessary pad the shorter strings with blanks on the right. For example,

```
nameAndAddress = ['Adam B Carr  '; '21 Barkly Ave';
                  'Discovery    ']
```

results in

```
nameAndAddress =

Adam B Carr
21 Barkly Ave
Discovery
```

An easier way to create two-dimensional strings is to use the `char` function which automatically pads the shorter strings for you:

```
nameAndAddress = char('Adam B Carr', '21 Barkly Ave',
                      'Discovery')
```

Remember that if you use single-subscript notation the array is referenced by column, so

```
nameAndAddress(1:3)
```

returns

```
A2D
```

(the first letter of each row).

6.6 *`eval` and text macros

If a MATLAB expression is 'encoded' as a string in a variable `t`, say, the function `eval(t)` causes the expression in `t` to be evaluated. (Strictly, the string is *interpreted*.) This is called a *text macro* facility. For example, if the assignment

```
s = 'x = -b/(2*a);'
```

is made, `eval(s)` makes MATLAB interpret the text in `s` as the statement

```
x = -b/(2*a);
```

which is then carried out with the current values of `a`, `b` and `c`.

Another use of `eval` could be as a 'user-friendly' way of inputting a function from the Command Window. Consider the following script:

```
f = input( 'Enter function (of x) to be plotted: ', 's');
x = 0:0.01:10;
plot(x, eval(f)),grid
```

Here's how the command line looks after you enter the function:

```
>> Enter function (of x) to be plotted:
   exp(-0.5*x) .* sin(x)
```

Use of the second argument `'s'` for `input` means that you don't need to enclose the text in quotes. Whatever expression you enter will be plotted, as long as it's a function of *x*.

Note that you should use `feval` (Chapter 10: **Function handles**) rather than `eval` whenever possible, since `feval` is faster and code using it can be compiled with the MATLAB compiler.

6.6.1 Error trapping with `eval` and `lasterr`

The use of `eval` to evaluate an expression which is input by a user can misfire if the user enters an invalid expression; the script grinds to a halt. However, `eval` together with the function `lasterr` enables you to trap and correct errors like this without the script crashing.

The function `eval` can take a second (string) argument, representing an expression to be executed if an error is encountered in the first argument. This is demonstrated below with `lasterr`, which MATLAB sets to the error message corresponding to the last error encountered. `lasterr` can also be set to any string value. See `help` for more details.

As an example, the script below will continue running until you enter a valid expression.

```
lasterr('1');                       % true (not empty)
while ~isempty(lasterr)
  lasterr('');                      % false (empty)
  f = input( 'Enter expression: ', 's' );
  f = [f ';'];                      % suppress display when
                                      evaluated
  eval(f, 'disp(''Dum-dum!'')');    % trap error and display
                                      message
  if ~isempty(lasterr)
    lasterr
    fprintf( 'Please try again!\n' )
  end
end
```

(note the use of double apostrophes in the second (string) argument of `eval`).

6.6.2 `eval` with `try...catch`

Hahn, the original author of this book (first & second editions) is indebted to Dean Redelinghuys and Grant Grobbelaar for the following example. The example provides an alternative way of dealing with errors generated by `eval`.

The user is invited to define variables in the workspace by typing them in. The user's input is evaluated by `eval`. If an invalid assignment is entered, the script would normally crash. However, if an error occurs in the `try` clause of a `try...catch` statement, control is transferred to the `catch` clause, which deals with the error. The `try...catch` statement is embedded in a `while` loop which perseveres until the user eventually enters a valid assignment statement!

```
stopflag = 0;
while ~stopflag
   clc;                              % Clear the screen!
   disp('Your variables are:')
   whos
   a = input('Enter Variable definition or empty to quit:
             ','s');
   if isempty(a),
     stopflag = 1;
   else
     try
       eval([a ';']);                % Force no output to
                                       command window!
     catch
       disp('Invalid variable assignment statement. The
            error was:');
       disp(['   ', lasterr]);
       disp('Press a key to continue');
       pause
     end
   end;
end
```

Summary

Sections 6.1, 6.2, & 6.3

- A matrix is a 2-D array. Elements may be referenced in the conventional way with two subscripts. Alternatively, one subscript may be used, in which case the matrix is 'unwound' by columns.

- The colon operator may be used to represent all the rows or columns of a matrix, and also as a single subscript.
- The keyword `end` refers to the last row or column of a matrix.
- Use `repmat` to duplicate rows or columns of a matrix.
- Use the empty array `[ ]` to delete rows or columns of a matrix.
- Arrays may have more than two dimensions. In the case of a 3-D array the third subscript may be thought of as numbering pages, where each page contains a matrix defined by the first two subscripts.
- The matrix operations of multiplication and exponentiation are implemented with the matrix operators `*` and `^`.

Sections 6.4, 6.5, & 6.6

- A MATLAB string (of text) is an array, where each element represents one character in the string.
- Since a string is usually enclosed in apostrophes (single quotes), if the apostrophe itself is to be included in the string, it must be repeated.
- Strings may be concatenated (joined) with square brackets.
- Characters are represented by the numerical ASCII codes.

  `double` returns the ASCII codes for the characters in a string.

  `char` returns the string represented by the ASCII codes in an array.
- When strings (of equal length) are compared in a logical expression, the ASCII codes of the two strings are compared element by element, resulting in a logical array.
- `char` may be used to create two-dimensional character arrays.
- `eval` interprets its string argument as a MATLAB statement. This is the basis for text macros in MATLAB.

  `eval` can be used with `lasterr` and `try...catch` to trap and correct errors.

EXERCISES

Sections 6.1, 6.2, & 6.3

6.1 Set up any 3 × 3 matrix `a`. Write some command-line statements to perform the following operations on `a`:

(a) interchange columns 2 and 3;
(b) add a fourth column (of 0's);

(c) insert a row of 1's as the new second row of `a` (i.e. move the current second and third rows down);
(d) remove the second column.

Sections 6.4, 6.5, & 6.6

6.2 Write a function `deblank(s)` which removes all the blanks from the string `s`.

6.3 Write a function `toupper(s)` which translates the lowercase letters in the string `s` to uppercase.

6.4 Write a function `ispalin(s)` which returns 1 if the string `s` is a *palindrome*, or 0 if it is not. A palindrome is a sentence which reads the same backwards as forwards, such as

Reward a Toyota drawer

(someone who draws Toyotas, presumably), or Napoleon's classic lament,

Able was I ere I saw Elba

Assume there is no punctuation, remove all the blanks first, and convert all letters to lowercase, say.

6.5 A formula, called Zeller's Congruence, may be used to compute the day of the week, given the date (within a certain range of dates). The formula is

$$f = 1 + ([2.6m - 0.2] + k + y + [y/4] + [c/4] - 2c) \text{ modulo } 7,$$

where the square brackets denote the integer part, modulo 7 means the remainder when divided by 7, and

- m is the month number, with January and February taken as months 11 and 12 of the preceding year, so March is then month 1, and December month 10;
- k is the day of the month;
- c is the century number;
- y is the year in the century;
- $f = 1$ means Sunday, 2 means Monday, etc.

For example, 23rd August 1963 is represented by $m = 6$, $k = 23$, $c = 19$, $y = 63$; 1st January 1800 is represented by $m = 11$, $k = 1$, $c = 17$, $y = 99$.

Write a function `dayofweek(d)` which takes the date in the form of a vector `d = [dd mm yyyy]` (e.g. `[9 3 2001]` for 9 March 2001) and returns the day of the week (in words) on which it falls. Test your program on some known dates, like today's date, or your birthday, or 7 December 1941 (Pearl Harbor Sunday). **Hint:** Use a 2-D array of characters for the days of the week.

The formula will not work if you go too far back. Shakespeare and Cervantes both died on 23 April 1616. Shakespeare died on a Tuesday, but Cervantes died on a Saturday! This is because England had not yet adopted the Gregorian calendar and was consequently ten days behind the rest of the world. The formula will also not work if you go too far forward.

6.6 Write a script which will input a number in binary code (e.g. 1100—no blanks between the digits) and write its decimal value (12 in this case). **Hint:** Input the number as a string, and make use of the fact that the string is an array.

6.7 Write a function `rightfill(s, n)` which fills the string `s` with blanks from the right until its total length is `n` characters. If `s` is longer than `n` characters, it must be truncated.

6.8 Write a function `alphacomp(str1, str2)` which returns 1 if the string `str1` is ahead of `str2` alphabetically, and 0 otherwise.

The function should work even if the strings are not of equal length.

If you are feeling energetic, you might like to use `alphacomp` as the basis for sorting a list of words alphabetically.

7

Introduction to graphics

The objective of this chapter is to introduce you to

- **MATLAB's high-level 2-D and 3-D plotting facilities.**

A picture, it is said, is worth a thousand words. MATLAB has a powerful graphics system for presenting and visualizing data, which is reasonably easy to use. (Most of the figures in this book have been generated by MATLAB.)

This chapter introduces MATLAB's high-level 2-D and 3-D plotting facilities. Low-level features, such as handle graphics, are discussed later in this chapter.

It should be stressed that the treatment of graphics in this chapter is of necessity brief, and intended to give you a glimpse of the richness and power of MATLAB graphics. For the full treatment, you should consult `help` on the functions mentioned in this chapter, as well as the comprehensive list of graphics functions in the online Help, under **MATLAB: Reference: MATLAB Function Reference: Functions by Category: Graphics**.

7.1 Basic 2-D graphs

Graphs (in 2-D) are drawn with the `plot` statement. In its simplest form, it takes a single vector argument as in `plot(y)`. In this case the elements of `y` are plotted against their indexes, e.g. `plot(rand(1, 20))` plots 20 random numbers against the integers 1–20, joining successive points with straight lines, as in Figure 7.1(a). If `y` is a matrix, its columns are plotted against element indexes.

Axes are automatically scaled and drawn to include the minimum and maximum data points.

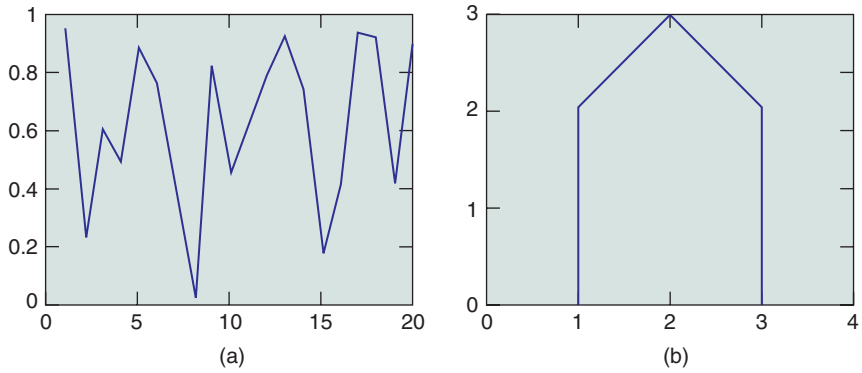


Figure 7.1 Examples of `plot`

Probably the most common form of `plot` is `plot(x, y)` where `x` and `y` are vectors of the same length, e.g.

```
x = 0:pi/40:4*pi;
plot(x, sin(x))
```

In this case, the coordinates of the *i*th point are x_i, y_i. This form of `plot` has been used widely in earlier chapters.

Straight-line graphs are drawn by giving the *x* and *y* coordinates of the end-points in two vectors. For example, to draw a line between the point with cartesian coordinates (0, 1) and (4, 3) use the statement

```
plot([0 4], [1 3])
```

i.e. `[0 4]` contains the *x* coordinates of the two points, and `[1 3]` contains their *y* coordinates.

MATLAB has a set of 'easy-to-use' plotting commands, all starting with the string '`ez`'. The easy-to-use form of `plot` is `ezplot`, e.g.

```
ezplot('tan(x)')
```

EXERCISES

1. Draw lines joining the following points: (0, 1), (4, 3), (2, 0) and (5, −2).
2. See if you can draw a 'house' similar to the one depicted in Figure 7.1(b).

7.1.1 Labels

Graphs may be labeled with the following statements:

`gtext('text')`
writes a string (`'text'`) in the graph window. `gtext` puts a cross-hair in the graph window and waits for a mouse button or keyboard key to be pressed. The cross-hair can be positioned with the mouse or the arrow keys. For example,

```
gtext( 'X marks the spot' )
```

Go on—try it!

Text may also be placed on a graph interactively with **Tools -> Edit Plot** from the figure window.

`grid`
adds/removes grid lines to/from the current graph. The grid state may be toggled.

`text(x, y, 'text')`
writes text in the graphics window at the point specified by `x` and `y`.

If `x` and `y` are vectors, the text is written at each point. If the text is an indexed list, successive points are labeled with corresponding rows of the text.

`title('text')`
writes the text as a title on top of the graph.

`xlabel('horizontal')`
labels the *x*-axis.

`ylabel('vertical')`
labels the *y*-axis.

7.1.2 Multiple plots on the same axes

There are at least three ways of drawing multiple plots on the same set of axes (which may however be rescaled if the new data falls outside the range of the previous data).

1. The easiest way is simply to use `hold` to keep the current plot on the axes. All subsequent plots are added to the axes until `hold` is released, either with `hold off`, or just `hold`, which toggles the hold state.
2. The second way is to use `plot` with multiple arguments, e.g.

```
plot(x1, y1, x2, y2, x3, y3, ... )
```

   plots the (vector) pairs (`x1`, `y1`), (`x2`, `y2`), etc. The advantage of this method is that the vector pairs may have different lengths. MATLAB automatically selects a different color for each pair.

   If you are plotting two graphs on the same axes you may find `plotyy` useful—it allows you to have independent *y*-axis labels on the left and the right, e.g.

```
plotyy(x,sin(x), x, 10*cos(x))
```

   (for `x` suitably defined).
3. The third way is to use the form

```
plot(x, y)
```

   where `x` and `y` may both be matrices, or where one may be a vector and one a matrix.

   If one of `x` or `y` is a matrix and the other is a vector, the rows or columns of the matrix are plotted against the vector, using a different color for each. Rows or columns of the matrix are selected depending on which have the same number of elements as the vector. If the matrix is square, columns are used.

   If `x` and `y` are both matrices of the same size, the columns of `x` are plotted against the columns of `y`.

   If `x` is not specified, as in `plot(y)`, where `y` is a matrix, the columns of `y` are plotted against the row index.

7.1.3 Line styles, markers and color

Line styles, markers and colors may be selected for a graph with a string argument to `plot`, e.g.

```
plot(x, y, '--')
```

joins the plotted points with dashed lines, whereas

```
plot(x, y, 'o')
```

draws circles at the data points with no lines joining them. You can specify all three properties, e.g.

```
plot(x,sin(x), x, cos(x), 'om--')
```

plots $\sin(x)$ in the default style and color and $\cos(x)$ with circles joined with dashes in magenta. The available colors are denoted by the symbols `c`, `m`, `y`, `k`, `r`, `g`, `b`, `w`. You can have fun trying to figure out what they mean, or you can use `help plot` to see the full range of possible symbols.

7.1.4 Axis limits

Whenever you draw a graph with MATLAB it automatically scales the axis limits to fit the data. You can override this with

```
axis( [xmin, xmax, ymin, ymax] )
```

which sets the scaling on the *current* plot, i.e. draw the graph first, then reset the axis limits.

If you want to specify one of the minimum or maximum of a set of axis limits, but want MATLAB to autoscale the other, use `Inf` or `-Inf` for the autoscaled limit.

You can return to the default of automatic axis scaling with

```
axis auto
```

The statement

```
v = axis
```

returns the current axis scaling in the vector `v`.

Scaling is frozen at the current limits with

```
axis manual
```

so that if `hold` is turned on, subsequent plots will use the same limits.

If you draw a circle, e.g. with the statements

```
x = 0:pi/40:2*pi;
plot(sin(x), cos(x))
```

the circle probably won't appear round, especially if you resize the figure window. The command

```
axis equal
```

makes unit increments along the *x*- and *y*-axis the same physical length on the monitor, so that circles always appear round. The effect is undone with `axis normal`.

You can turn axis labeling and tick marks off with `axis off`, and back on again with `axis on`.

axes and axis?

You and I might be forgiven for thinking that the word 'axes' denotes the plural of 'axis', which it does indeed in common English usage. However, in MATLAB the word 'axes' refers to a particular graphics *object*, which includes not only the *x*-axis and *y*-axis and their tick marks and labels, but also everything drawn on those particular axes: the actual graphs and any text included in the figure. Axes objects are discussed in more detail later in this chapter.

7.1.5 Multiple plots in a figure: `subplot`

You can show a number of plots in the same figure window with the `subplot` function. It looks a little curious at first, but it's quite easy to get the hang of it. The statement

```
subplot(m, n, p)
```

divides the figure window into $m \times n$ small sets of axes, and selects the pth set for the current plot (numbered by row from the left of the top row). For example, the following statements produce the four plots shown in Figure 7.2 (details of 3-D plotting are discussed in Section 7.2).

```
[x, y] = meshgrid(-3:0.3:3);
z = x .* exp(-x.^2 - y.^2);
```

```
subplot(2,2,1)
mesh(z),title('subplot(2,2,1)')
subplot(2,2,2)
mesh(z)
view(-37.5,70),title('subplot(2,2,2)')
subplot(2,2,3)
mesh(z)
view(37.5,-10),title('subplot(2,2,3)')
subplot(2,2,4)
mesh(z)
view(0,0),title('subplot(2,2,4)')
```

The command `subplot(1,1,1)` goes back to a single set of axes in the figure.

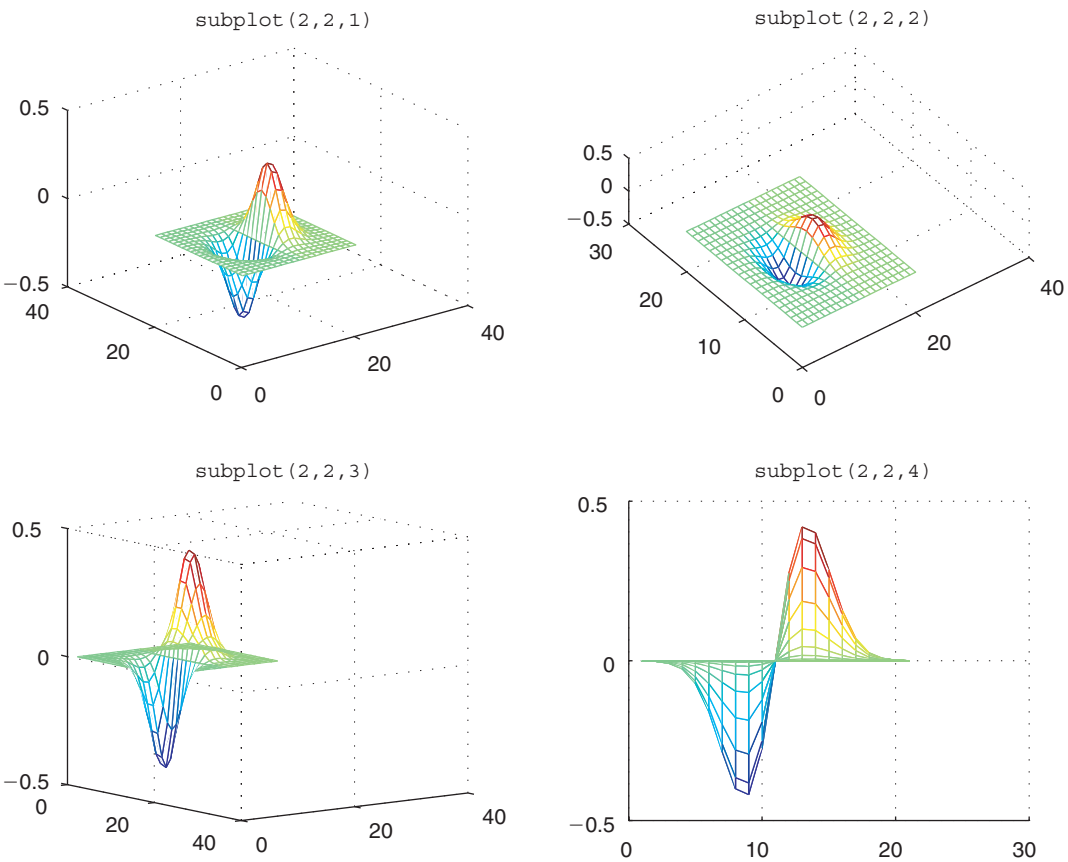


Figure 7.2 Four subplots: rotations of a 3-D surface

7.1.6 `figure`, `clf` and `cla`

`figure(h)`, where `h` is an integer, creates a new figure window, or makes figure `h` the current figure. Subsequent plots are drawn in the current figure. `h` is called the *handle* of the figure. Handle graphics is discussed further in a later section of this chapter.

`clf` clears the current figure window. It also resets all properties associated with the axes, such as the hold state and the axis state.

`cla` deletes all plots and text from the current axes, i.e. leaves only the *x*- and *y*-axes and their associated information.

7.1.7 Graphical input

The command

```
[x, y] = ginput
```

allows you to select an unlimited number of points from the current graph using a mouse or arrow keys. A movable cross-hair appears on the graph. Clicking saves its coordinates in `x(i)` and `y(i)`. Pressing **Enter** terminates the input. An example is provided in the last chapter of this book which involves selecting points in a figure and fitting a curve to them.

The command

```
[x, y] = ginput(n)
```

works like `ginput` except that you must select exactly `n` points.

See `help` for further information.

7.1.8 Logarithmic plots

The command

```
semilogy(x, y)
```

plots `y` with a $\log_{10}$ scale and `x` with a linear scale. For example, the statements

```
x = 0:.01:4;
semilogy(x, exp(x)), grid
```

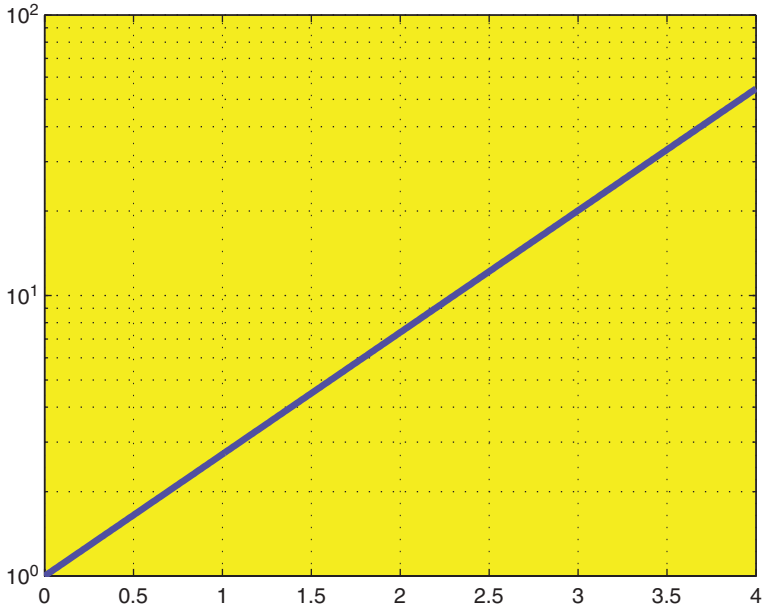


Figure 7.3 A logarithmic plot

produce the graph in Figure 7.3. Equal increments along the *y*-axis represent multiples of powers of 10. So, starting from the bottom, the grid lines are drawn at $1, 2, 3, \ldots, 10, 20, 30 \ldots, 100, \ldots$. Incidentally, the graph of e^x on these axes is a straight line, because the equation $y = e^x$ transforms into a linear equation when you take logs of both sides.

See also `semilogx` and `loglog`.

Note that `x` and `y` may be vectors and/or matrices, just as in `plot`.

EXERCISE

Draw graphs of x^2, x^3, x^4 and e^{x^2} over the interval $0 \leq x \leq 4$, using `semilogy`.

7.1.9 Polar plots

The point (x, y) in cartesian coordinates is represented by the point (θ, r) in *polar* coordinates, where

$$x = r\cos(\theta),$$

$$y = r\sin(\theta),$$

and θ varies between 0 and 2π radians (360°).

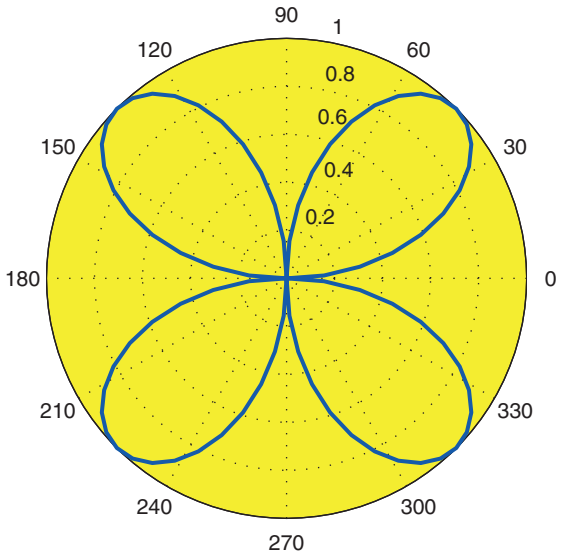


Figure 7.4 Polar plot of θ against $\sin(2\theta)$

The command

```
polar(theta, r)
```

generates a polar plot of the points with angles in `theta` and magnitudes in `r`.

As an example, the statements

```
x = 0:pi/40:2*pi;
polar(x, sin(2*x)),grid
```

produce the plot shown in Figure 7.4.

7.1.10 Plotting rapidly changing mathematical functions: `fplot`

In all the graphing examples so far, the *x* coordinates of the points plotted have been incremented uniformly, e.g. `x = 0:0.01:4` . If the function being plotted changes very rapidly in some places, this can be inefficient, and can even give a misleading graph.

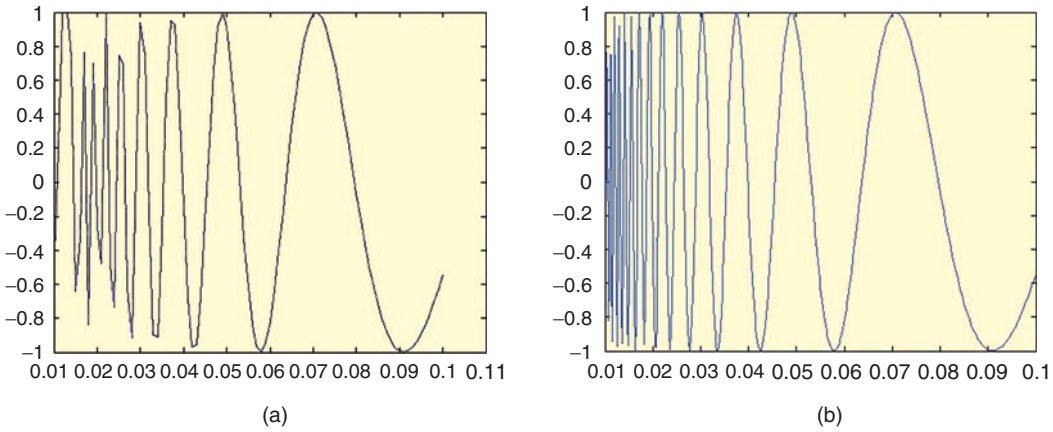


Figure 7.5 $y = \sin(1/x)$

For example, the statements

```
x = 0.01:.001:.1;
plot(x, sin(1./x))
```

produce the graph shown in Figure 7.5(a). But if the `x` increments are reduced to 0.0001, we get the graph in Figure 7.5(b) instead. For $x < 0.04$, the two graphs look quite different.

MATLAB has a function called `fplot` which uses a more elegant approach. Whereas the above method evaluates $\sin(1/x)$ at equally spaced intervals, `fplot` evaluates it more frequently over regions where it changes more rapidly. Here's how to use it:

```
fplot('sin(1/x)', [0.01 0.1]) % no, 1./x not needed!
```

7.1.11 The property editor

The most general way of editing a graph is by using the Property Editor, e.g. **Edit - > Figure Properties** from the figure window. This topic is discussed briefly towards the end of this chapter.

7.2 3-D plots

MATLAB has a variety of functions for displaying and visualizing data in 3-D, either as lines in 3-D, or as various types of surfaces. This section provides a brief overview.

7.2.1 plot3

The function `plot3` is the 3-D version of `plot`. The command

```
plot3(x, y, z)
```

draws a 2-D projection of a line in 3-D through the points whose coordinates are the elements of the vectors `x`, `y` and `z`. For example, the command

```
plot3(rand(1,10), rand(1,10), rand(1,10))
```

generates 10 random points in 3-D space, and joins them with lines, as shown in Figure 7.6(a).

As another example, the statements

```
t = 0:pi/50:10*pi;
plot3(exp(-0.02*t).*sin(t), exp(-0.02*t).*cos(t), ...
xlabel('x-axis'), ylabel('y-axis'), zlabel('z-axis')
```

produce the inwardly spiraling helix shown in Figure 7.6(b). Note the orientation of the *x*-, *y*- and *z*-axes, and in particular that the *z*-axis may be labeled with `zlabel`.

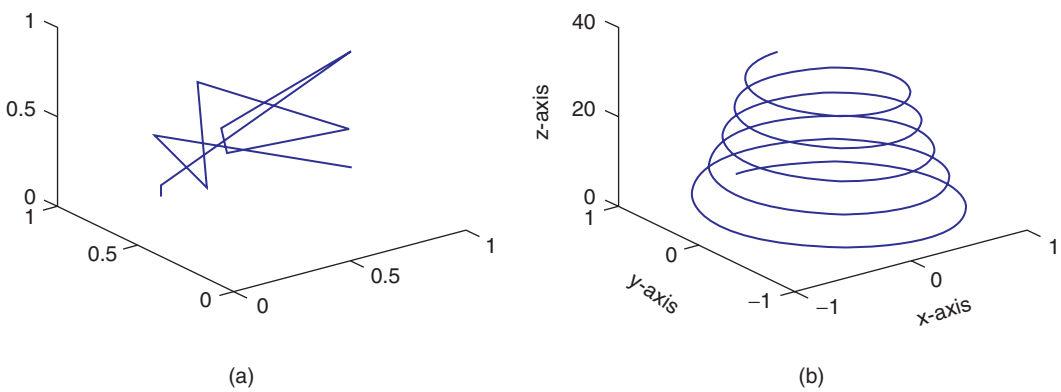


Figure 7.6 Examples of `plot3`

7.2.2 Animated 3-D plots with `comet3`

The function `comet3` is similar to `plot3` except that it draws with a moving 'comet head.' Use `comet3` to animate the helix in Figure 7.6(b). No prizes for guessing the name of the 2-D version.

7.2.3 Mesh surfaces

In Chapter 1 we saw how to draw the Mexican hat (Figure 1.3):

```
[x y ] = meshgrid(-8 : 0.5 : 8);
r = sqrt(x.^2 + y.^2) + eps;
z = sin(r) ./ r;
```

This drawing is an example of a *mesh surface*.

To see how such surface is drawn, let's take a simpler example, say $z = x^2 - y^2$. The surface we are after is the one generated by the values of z as we move around the x-y plane. Let's restrict ourselves to part of the first quadrant of this plane, given by

$$0 \leq x \leq 5, \quad 0 \leq y \leq 5.$$

The first step is to set up the *grid* in the x-y plane over which the surface is to be plotted. You can use the MATLAB function `meshgrid` to do it, as follows:

```
[x y] = meshgrid(0:5);
```

This statement sets up two matrices, `x` and `y`. (Functions, such as `meshgrid`, which return more than one 'output argument', are discussed in detail in Chapter 10. However, you don't need to know the details in order to be able to use it here.)

The two matrices in this example are:

```
x =
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
```

```
y =
     0     0     0     0     0     0
     1     1     1     1     1     1
     2     2     2     2     2     2
     3     3     3     3     3     3
     4     4     4     4     4     4
     5     5     5     5     5     5
```

The effect of this is that the *columns* of the matrix `x` as it is displayed hold the *x* coordinates of the points in the grid, while the *rows* of the display of `y` hold the *y* coordinates. Recalling the way MATLAB array operations are defined, element by element, this means that the statement

```
z = x.^2 - y.^2
```

will correctly generate the surface points:

```
z =
     0     1     4     9    16    25
    -1     0     3     8    15    24
    -4    -3     0     5    12    21
    -9    -8    -5     0     7    16
   -16   -15   -12    -7     0     9
   -25   -24   -21   -16    -9     0
```

For example, at the grid point (5, 2), *z* has the value $5^2 - 2^2 = 21$. Incidentally, you don't need to worry about the exact relationship between grid coordinates and matrix subscripts; this is taken care of by `meshgrid`.

The statement `mesh(z)` then plots the surface (Figure 7.7), with mesh lines connecting the points in the surface that lie above grid points.

Note that `mesh(z)` shows the row and column indices (subscripts) of the matrix `z` on the *x* and *y* axes. If you want to see proper values on the *x* and *y* axes use `mesh(x,y,z)`. This applies to many of the other 3-D plotting functions.

The function `mesh` draws a surface as a 'wire frame'. An alternative visualization is provided by `surf`, which generates a faceted view of the surface (in color), i.e. the wire frame is covered with small tiles.

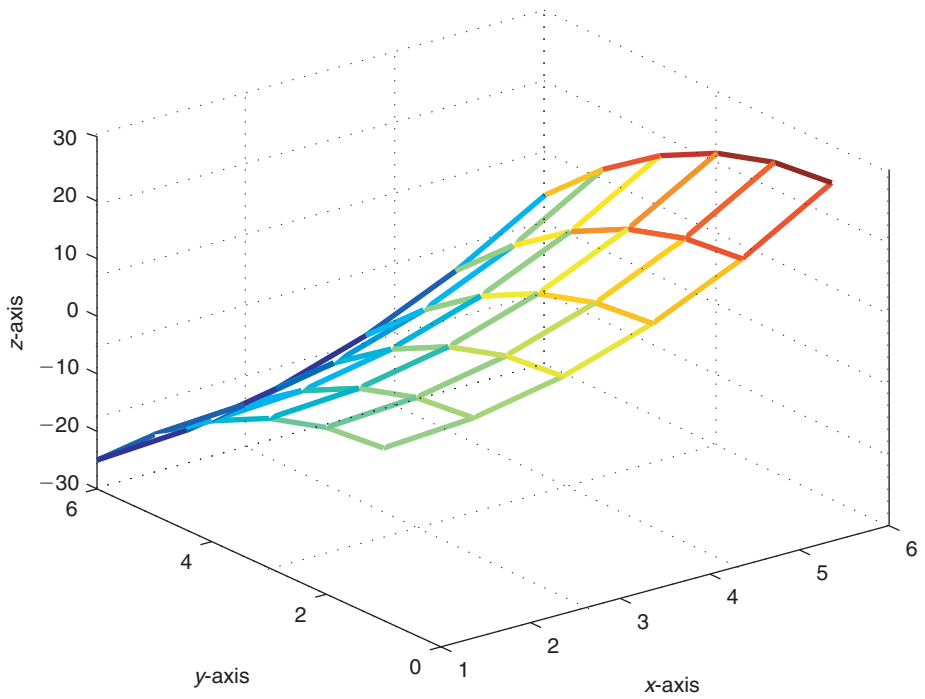


Figure 7.7 The surface $z = x^2 - y^2$

See `help` for variations on `mesh` and `surf`.

EXERCISES

1. Draw the surface shown in Figure 7.7 with a finer mesh (of 0.25 units in each direction), using

```
[x y] = meshgrid(0:0.25:5);
```

(the number of mesh points in each direction is 21).

2. The initial heat distribution over a steel plate is given by the function

$$u(x, y) = 80y^2 e^{-x^2 - 0.3y^2}.$$

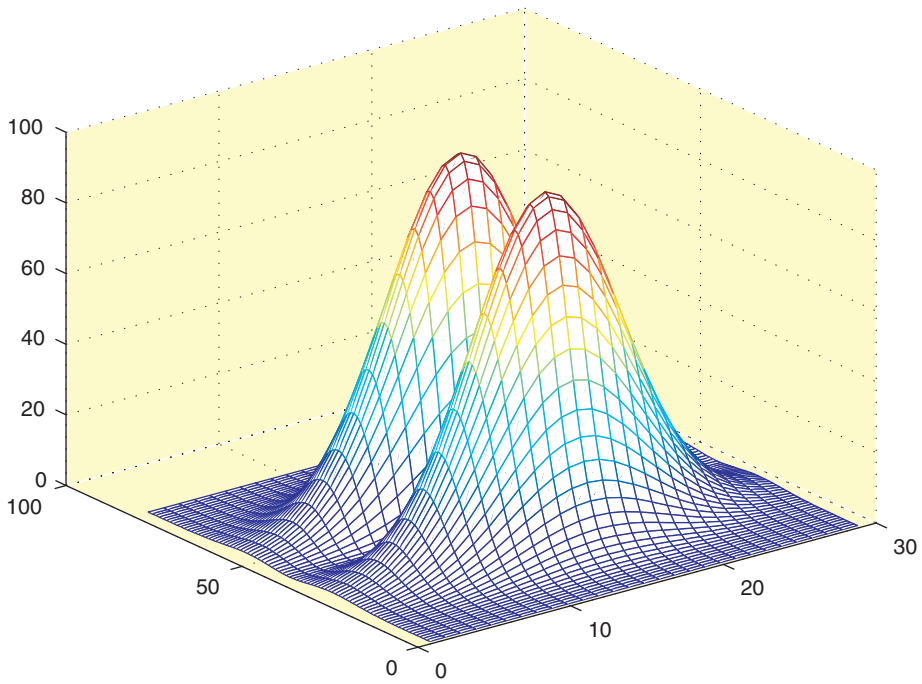


Figure 7.8 Heat distribution over a steel plate

Plot the surface u over the grid defined by

$$-2.1 \leq x \leq 2.1, \quad -6 \leq y \leq 6,$$

where the grid width is 0.15 in both directions. You should get the plot shown in Figure 7.8.

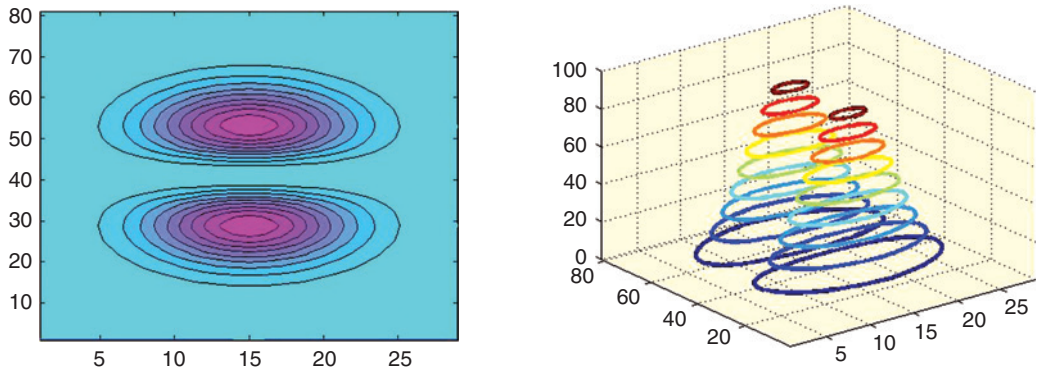
7.2.4 Contour plots

If you managed to draw the plot in Figure 7.8, try the command

```
contour(u)
```

You should get a *contour plot* of the heat distribution, as shown in Figure 7.9(a), i.e. the *isothermals* (lines of equal temperature). Here's the code:

```
[x, y] = meshgrid(-2.1:0.15:2.1, -6:0.15:6);
u = 80 * y.^2 .* exp(-x.^2 - 0.3*y.^2);
contour(u)
```



(a) (b)

Figure 7.9 Contour plots

The function `contour` can take a second input variable. It can be a scalar specifying how many contour levels to plot, or it can be a vector specifying the values at which to plot the contour levels.

You can get a 3-D contour plot with `contour3(u)`, as shown in Figure 7.9(b).

Contour levels may be labeled with `clabel` (see `help`).

A 3-D contour plot may be drawn under a surface with `meshc` or `surfc`. For example, the statements

```
[x y] = meshgrid(-2:.2:2);
z = x .* exp(-x.^2 - y.^2);
meshc(z)
```

produce the graph in Figure 7.10(a).

7.2.5 Cropping a surface with `NaNs`

If a matrix for a surface plot contains `NaNs`, these elements are not plotted. This enables you to cut away (crop) parts of a surface. For example, the statements

```
[x y] = meshgrid(-2:.2:2, -2:.2:2);
z = x .* exp(-x.^2 - y.^2);
c = z;        % preserve the original surface
c(1:11,1:21) = nan*c(1:11,1:21);
mesh(c), xlabel('x-axis'), ylabel('y-axis')
```

produce the graph in Figure 7.10(b).

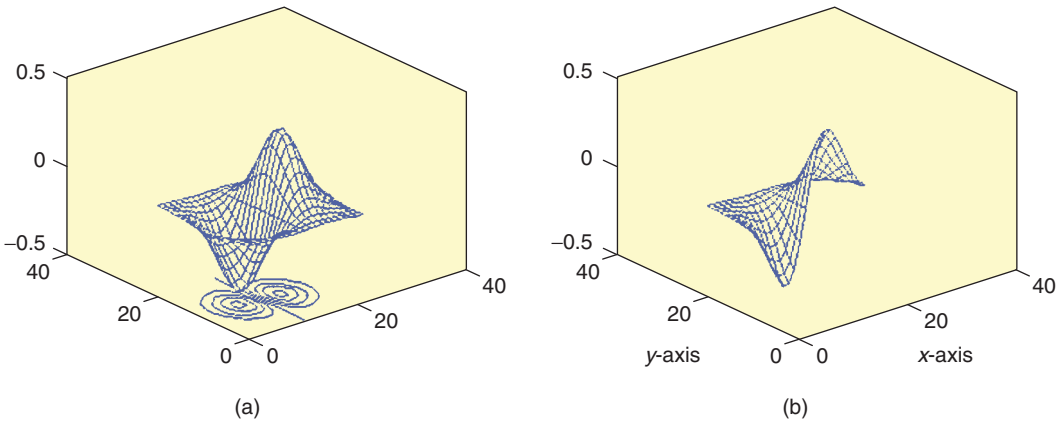


Figure 7.10 (a) `meshc`; (b) cropping a surface

7.2.6 Visualizing vector fields

The function `quiver` draws little arrows to indicate a gradient or other vector field. Although it produces a 2-D plot, it's often used in conjunction with `contour`, which is why it's described briefly here.

As an example, consider the scalar function of two variables $V = x^2 + y$. The *gradient* of V is defined as the *vector* field

$$\nabla V = \left(\frac{\partial V}{\partial x}, \frac{\partial V}{\partial y} \right)$$
$$= (2x, 1).$$

The following statements draw arrows indicating the direction of ∇V at points in the x-y plane (see Figure 7.11):

```
[x y] = meshgrid(-2:.2:2, -2:.2:2);
V = x.^2 + y;
dx = 2*x;
dy = dx;            % dy same size as dx
dy(:,:) = 1;        % now dy is same size as dx but all 1's
contour(x, y, V), hold on
quiver(x, y, dx, dy), hold off
```

The 'contour' lines indicate families of *level surfaces*; the gradient at any point is perpendicular to the level surface which passes through that point. The vectors `x` and `y` are needed in the call to `contour` to specify the axes for the contour plot.

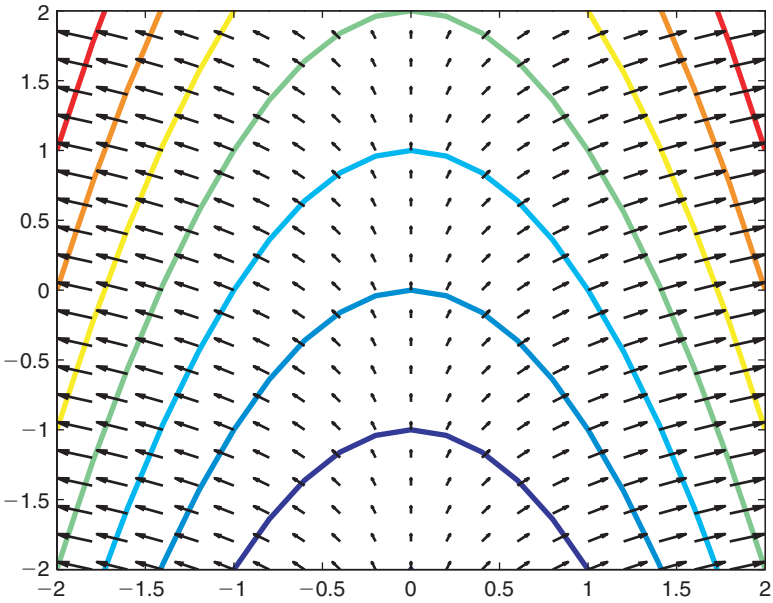


Figure 7.11 Gradients and level surfaces

An additional optional argument for `quiver` specifies the length of the arrows. See `help`.

If you can't (or don't want to) differentiate *V*, you can use the `gradient` function to estimate the derivative:

```
[dx dy] = gradient(V, 0.2, 0.2);
```

The values 0.2 are the increments in the *x* and *y* directions used in the approximation.

7.2.7 Visualization of matrices

The `mesh` function can also be used to 'visualize' a matrix. The following statements generate the plot in Figure 7.12:

```
a = zeros(30,30);
a(:,15) = 0.2*ones(30,1);
a(7,:) = 0.1*ones(1,30);
a(15,15) = 1;
mesh(a)
```

The matrix `a` is 30 × 30. The element in the middle—`a(15,15)`—is 1, all the elements in row 7 are 0.1, and all the remaining elements in column 15 are 0.2. `mesh(a)` then interprets the rows and columns of `a` as an *x*-*y* coordinate grid, with the values `a(i,j)` forming the mesh surface above the points (`i`, `j`).

The function `spy` is useful for visualizing sparse matrices.

7.2.8 Rotation of 3-D graphs

The `view` function enables you to specify the angle from which you view a 3-D graph. To see it in operation, run the following program, which rotates the visualized matrix in Figure 7.12:

```
a = zeros(30,30);
a(:,15) = 0.2*ones(30,1);
a(7,:) = 0.1*ones(1,30);
a(15,15) = 1;
mesh(a)
```

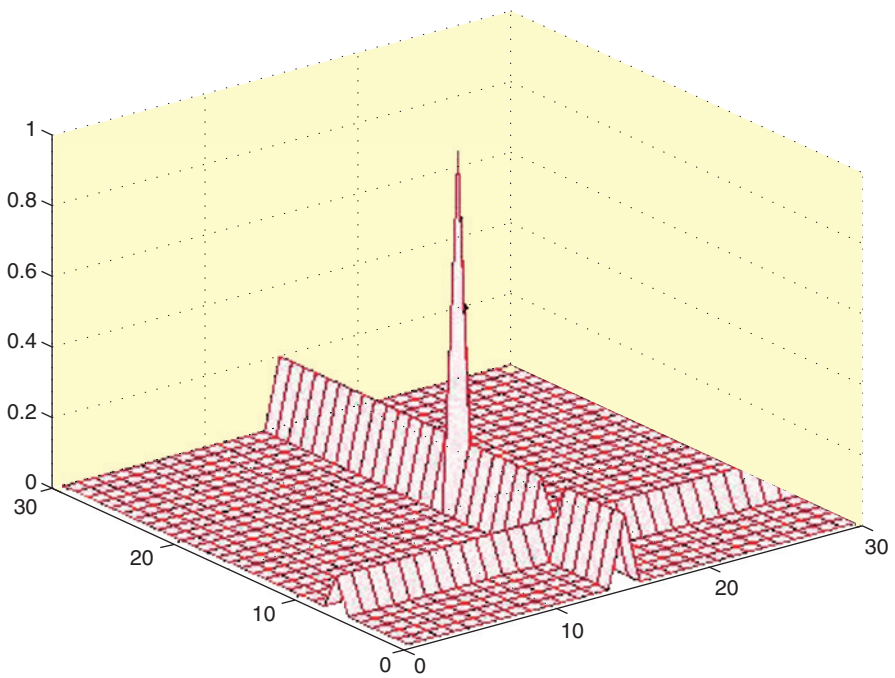


Figure 7.12 Visualization of a matrix

```
for az = -37.5:15:-37.5+360
  mesh(a), view(az, el)
  pause(0.5)
end
```

The function `view` takes two arguments. The first one, `az` in this example, is called the *azimuth* or polar angle in the *x*-*y* plane (in degrees). `az` rotates the *viewpoint* (you) about the *z*-axis—i.e. about the 'pinnacle' at (15,15) in Figure 7.12—in a counter-clockwise direction. The default value of `az` is $-37.5°$. The program therefore rotates you in a counter-clockwise direction about the *z*-axis in $15°$ steps starting at the default position.

The second argument of `view` is the vertical elevation `el` (in degrees). This is the angle a line from the viewpoint makes with the *x*-*y* plane. A value of $90°$ for `el` means you are directly overhead. Positive values of the elevation mean you are above the *x*-*y* plane; negative values mean you are below it. The default value of `el` is $30°$.

The command `pause(n)` suspends execution for `n` seconds.

You can rotate a 3-D figure interactively as follows. Click the Rotate 3-D button in the figure toolbar (first button from the right). Click on the axes and an outline of the figure appears to help you visualize the rotation. Drag the mouse in the direction you want to rotate. When you release the mouse button the rotated figure is redrawn.

EXERCISE

Rewrite the above program to change the elevation gradually, keeping the azimuth fixed at its default.

7.2.9 Other cool graphics functions

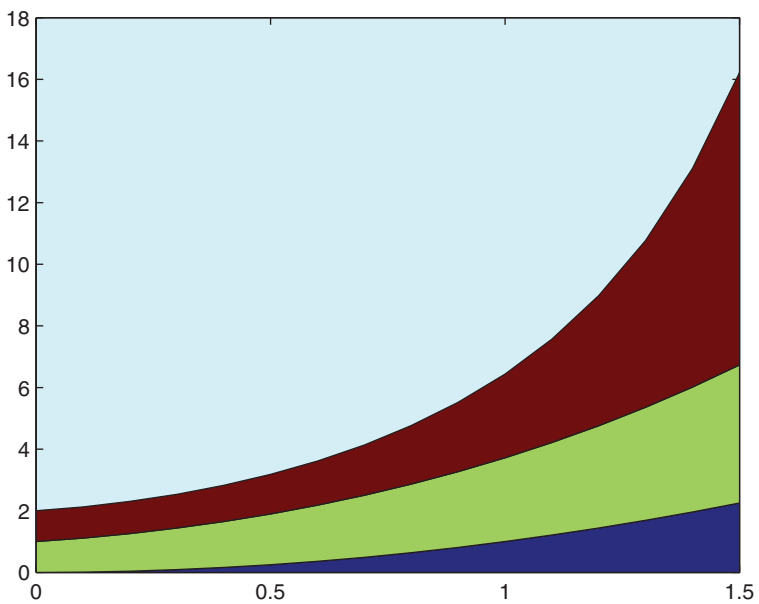
Here are more examples of interesting graphics functions. With each function a sample script and graph is shown. The list is by no means exhaustive and is meant to whet your appetite for more! You are encouraged to consult the list of graphics functions in the online **MATLAB FUNCTION REFERENCE**. Each entry has excellent examples.

Note that the MATLAB function `peaks` generates a sample surface (see under `meshz` below).

For reference purposes the figures below are all considered part of Figure 7.13. Some figures have been edited to render colors in gray-scale.

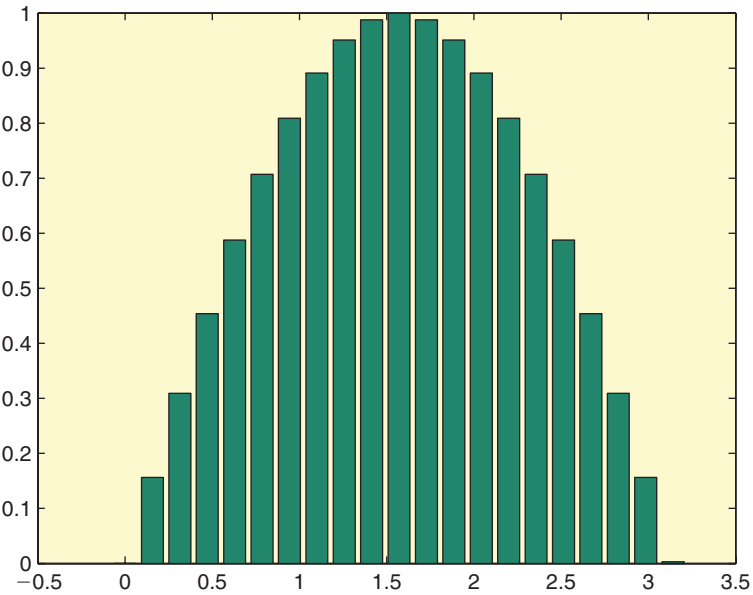
(a) **`area`**

```
x = 0:0.1:1.5;
area(x', [x.^2' ...
     exp(x)' exp(x.^2)'])
```



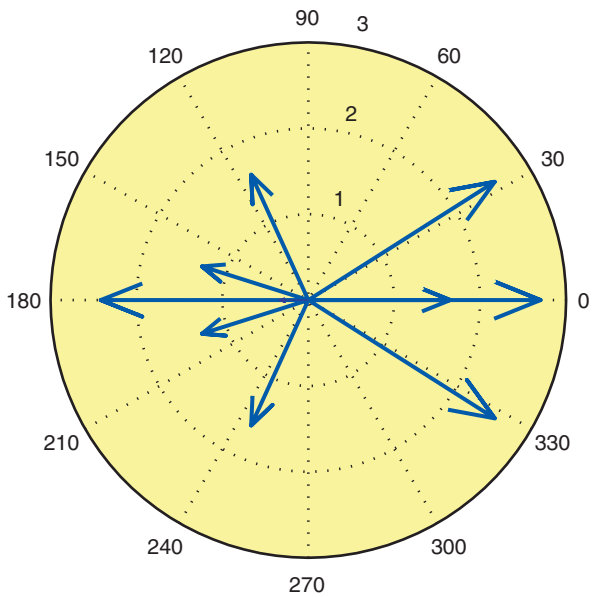
(b) **bar**

```
x = 0:pi/20:pi;
bar(x,sin(x))
```



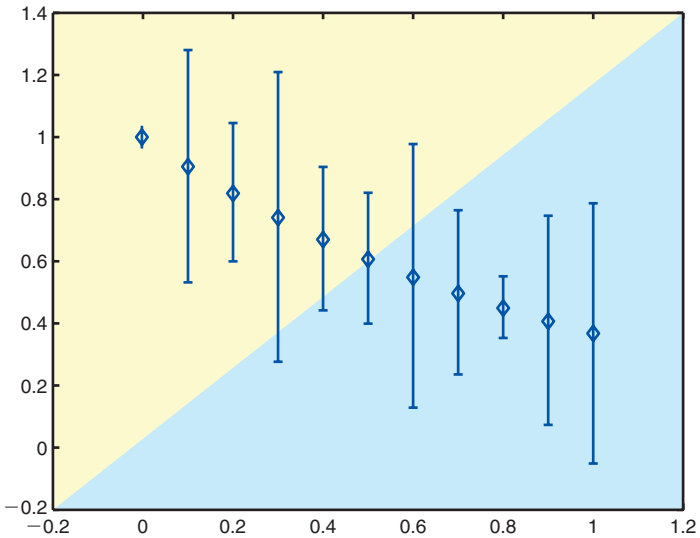
(c) **compass**

```
z = eig(randn(10));
compass(z)
```



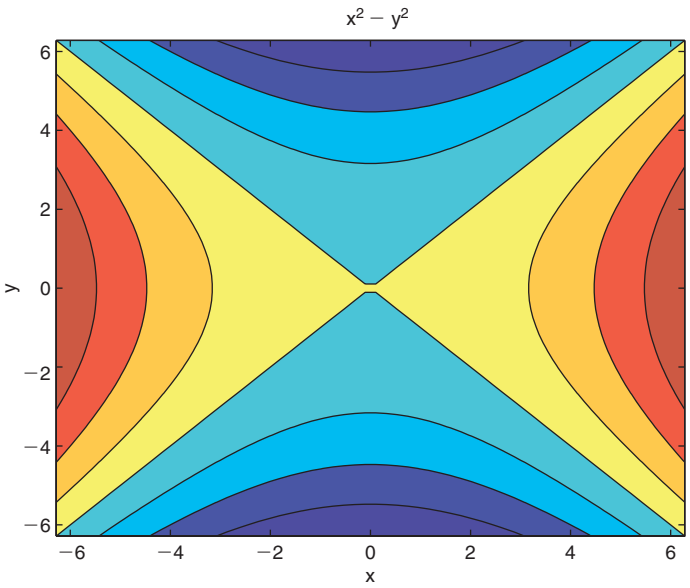
(d) **errorbar**

```
x = 0:0.1:1;
errorbar(x,exp(-x), 0.5*rand(1,length(x)),'d')
```



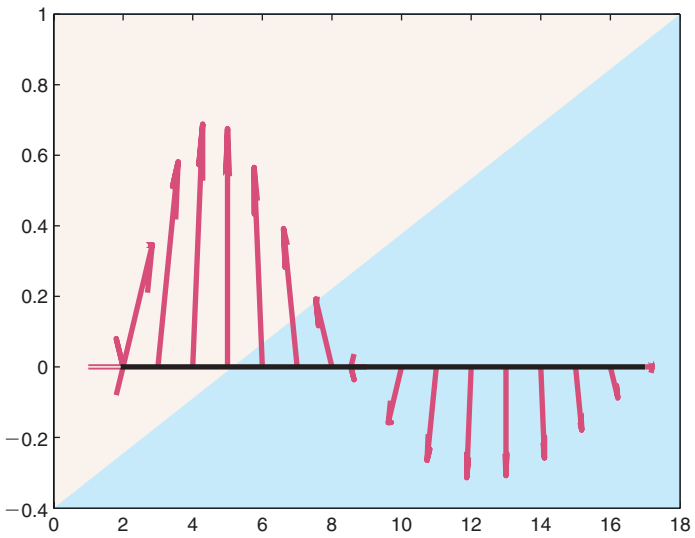
(e) **ezcontourf**

```
ezcontourf('x^2-y^2')
```



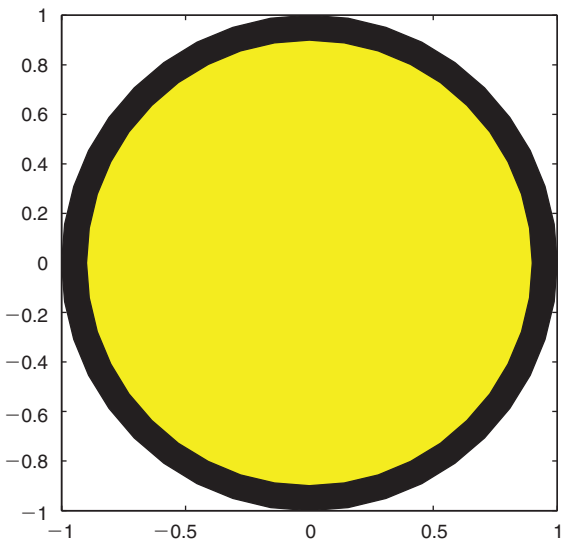
(f) **feather**

```
th = 0:pi/16:pi;
z = exp((-0.5+2*i)*th);
feather(z)
```



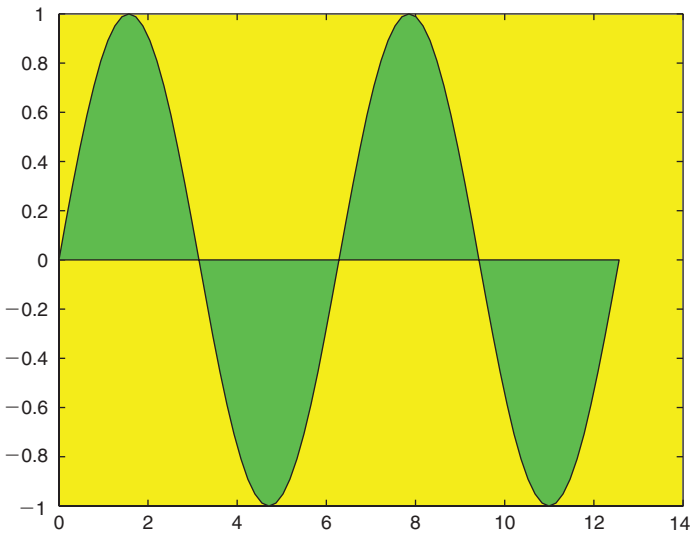
(g) **fill**

```
t = 0:pi/20:2*pi;
fill(cos(t),sin(t),'k', 0.9*cos(t),0.9*sin(t),'y'),
     axis square
     clear
```



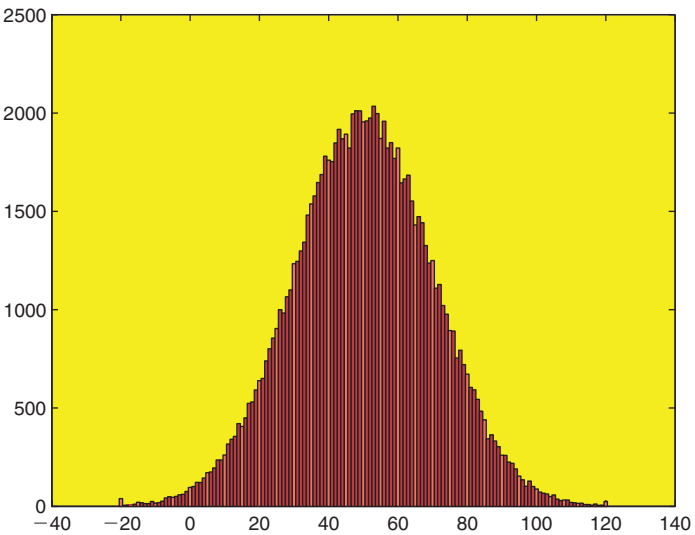
(g) fill (cont)

```
t = 0:pi/20:4*pi;
fill(t,sin(t),'g')
```



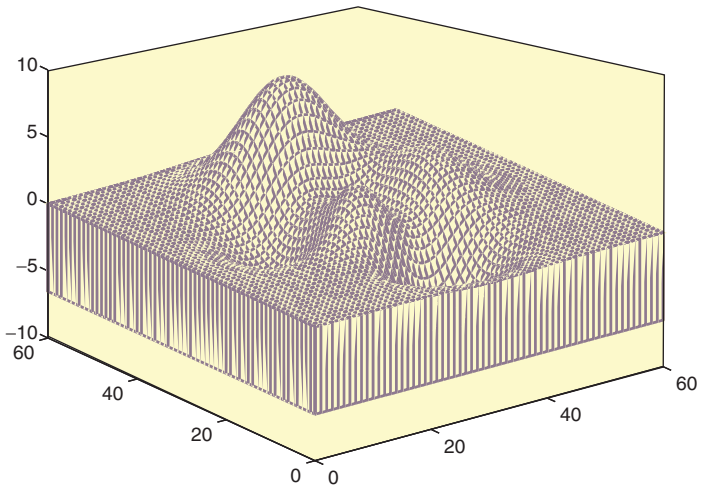
(h) hist

```
x = -20:120;
y = 50+20*randn(1,100000);
hist(y,x)
```



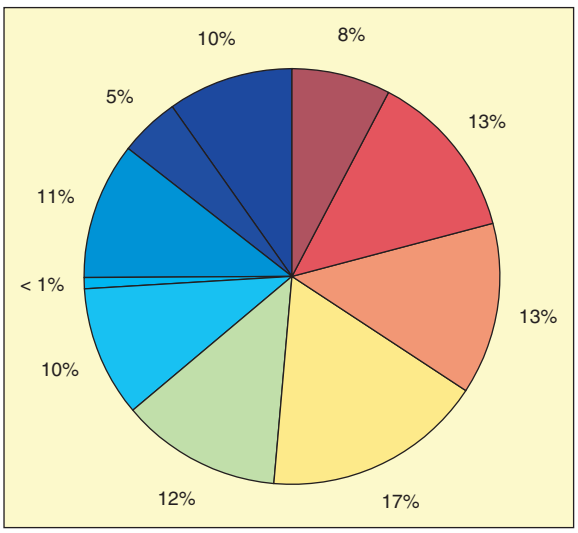
(i) meshz

```
[x y] = meshgrid(-3:0.1:3);
z = peaks(x,y);
meshz(z)
```



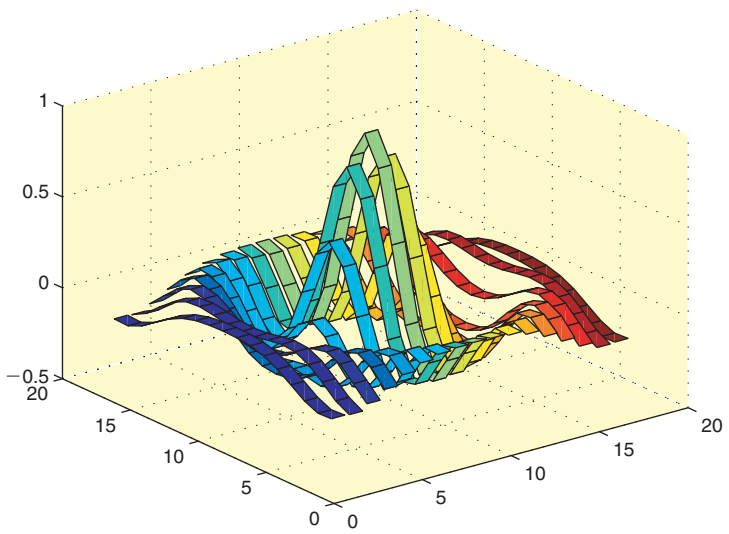
(j) pie

```
pie(rand(1,10))
```

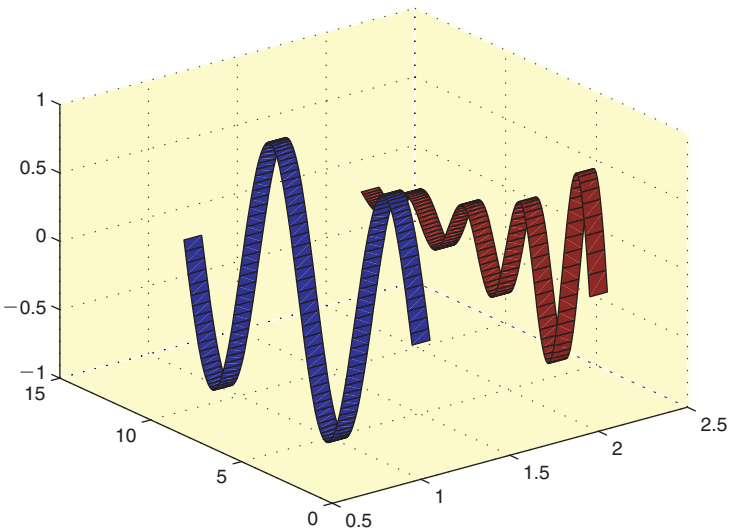


(k) ribbon

```
[x y] = meshgrid(-8 : 1 : 8);
r = sqrt(x.^2 + y.^2) + eps;
z = sin(r) ./ r;
ribbon(z)
```

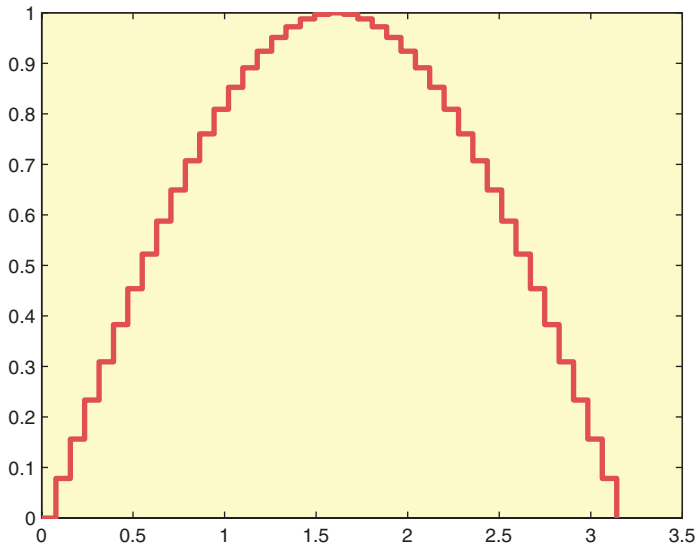



```
t = 0:pi/40:4*pi;
y1 = sin(t);
y2 = exp(-0.2*t).*sin(2*t);
y = [y1; y2];
ribbon(t', y', 0.1)
```



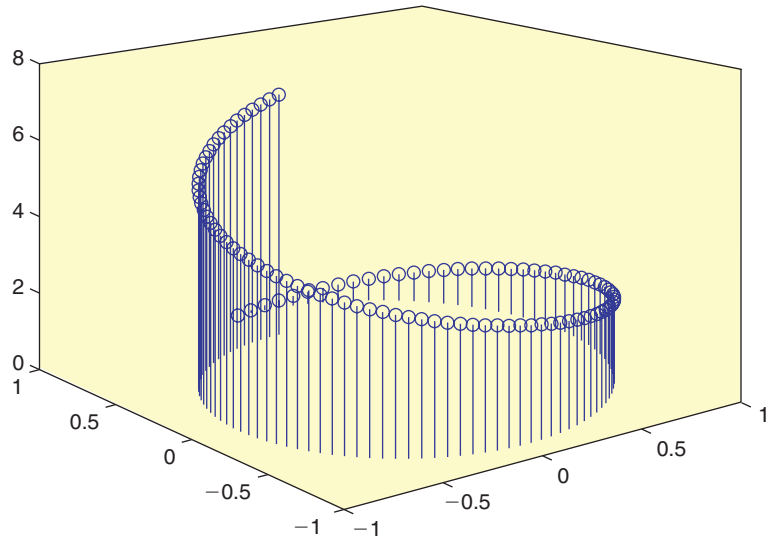
(l) **stairs**

```
x = 0:pi/40:pi;
stairs(x,sin(x))
```



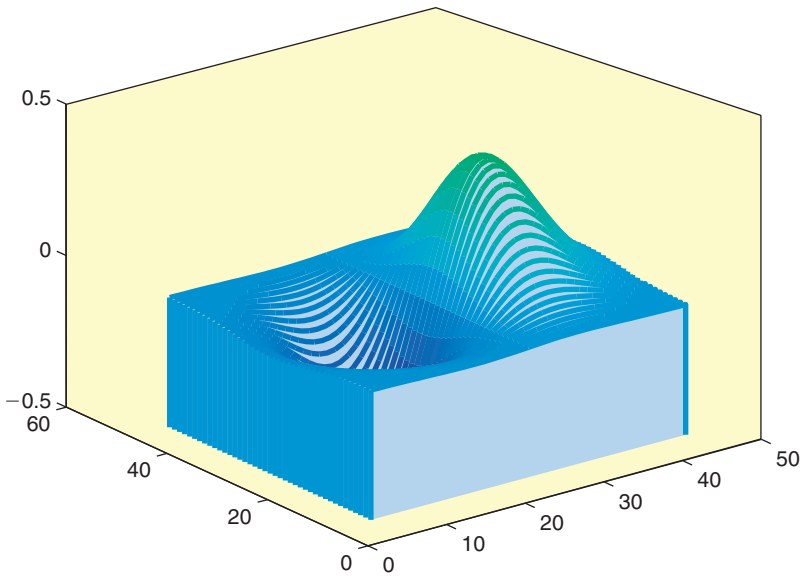
(m) **stem3**

```
t = 0:pi/50:2*pi;
r = exp(-0.05*t);
stem3(r.*sin(t), r.*cos(t),t)
```



(n) **waterfall**

```
[x y] = meshgrid(-2:0.1:2);
z = x.*exp(-x.^2-y.^2);
waterfall(z)
```



Summary

- 2-D plots are drawn with the `plot` statement.
- There is a set of easy-to-use plotters called `ez*`.
- Graphs may be labeled with `grid`, `text`, `title`, `xlabel`, `ylabel`, etc.
- Multiple plots may be drawn on the same axes in a number of ways.
- Line styles, markers and color may be varied.
- Axis limits may be set explicitly.
- In MATLAB the word 'axes' refers to the graphics object in which the *x* and *y* axis and their labels, plots and text annotations are drawn.
- A number of axes may be drawn in the same figure with `subplot`.
- Coordinates of points in a figure may be selected with `ginput`.
- `semilogx`, `semilogy` and `loglog` are used to plot on $\log_{10}$ scales.

- The `polar` command plots in polar coordinates.
- `fplot` provides a handy way of plotting mathematical functions.
- `plot3` draws lines in 3-D.
- `comet3` animates a 3-D plot.
- A 3-D surface may be plotted with `mesh`.
- A 3-D plot may be rotated with `view` or with the **Rotate 3-D** tool in the figure window.
- `mesh` may also be used to visualize a matrix.
- `contour` and `countour3` draws contour levels in 2-D and 3-D respectively.
- 3-D surfaces may be cropped.
- For a complete list of graphics functions consult the online Help in **MATLAB Function Reference: Functions by Category: Graphics**.

EXERCISES

7.1 Draw a graph of the population of the USA from 1790 to 2000, using the (logistic) model

$$P(t) = \frac{19\,72\,73\,000}{1 + e^{-0.03134(t-1913.25)}}$$

where t is the date in years.

Actual data (in 1000s) for every decade from 1790 to 1950 are as follows: 3929, 5308, 7240, 9638, 12 866, 17 069, 23 192, 31 443, 38 558, 50 156, 62 948, 75 995, 91 972, 105 711, 122 775, 131 669, 150 697. Superimpose this data on the graph of $P(t)$. Plot the data as discrete circles (i.e. do not join them with lines) as shown in Figure 7.14.

7.2 The Spiral of Archimedes (Figure 7.15(a)) may be represented in polar coordinates by the equation

$$r = a\theta,$$

where a is some constant. (The shells of a class of animals called nummulites grow in this way.) Write some command-line statements to draw the spiral for some values of a.

7.3 Another type of spiral is the *logarithmic* spiral (Figure 7.15(b)), which describes the growth of shells of animals like the periwinkle and the nautilus. Its equation

in polar coordinates is

$$r = aq^{\theta},$$

where $a > 0$, $q > 1$. Draw this spiral.

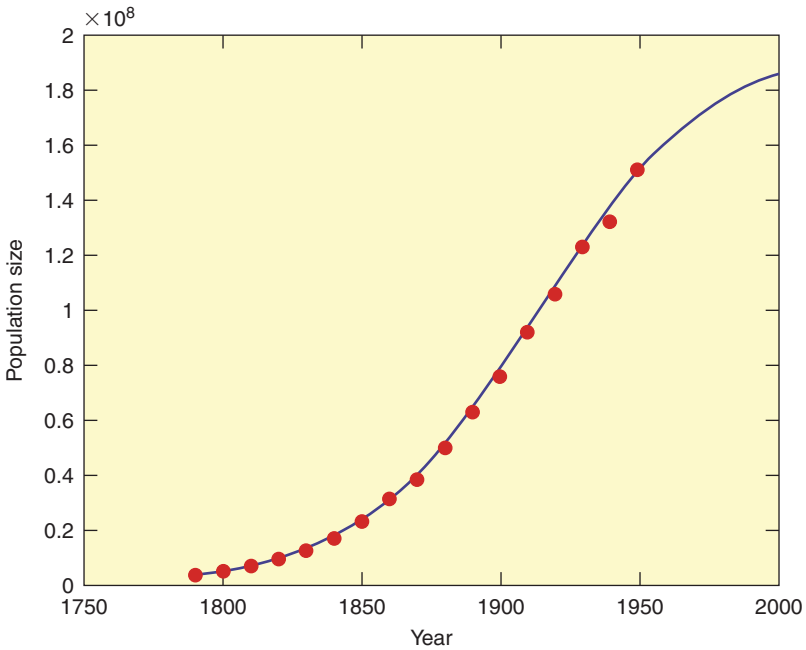


Figure 7.14 USA population: model and census data (○)

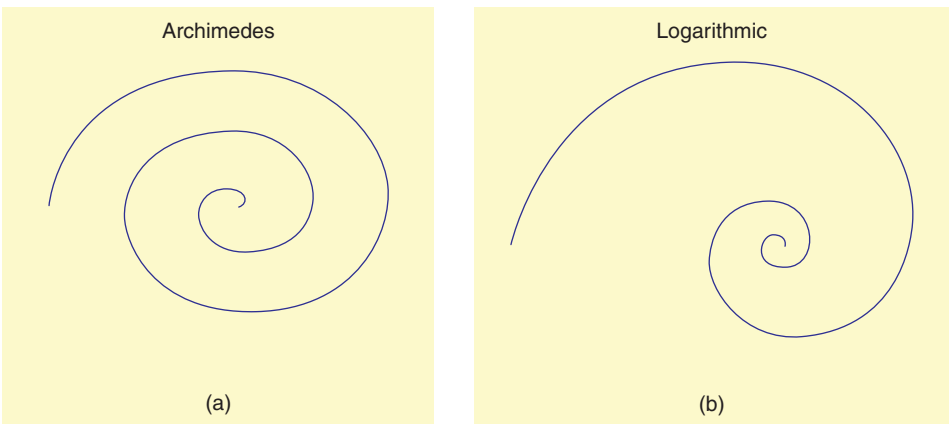


Figure 7.15 Spirals

7.4 The arrangement of seeds in a sunflower head (and other flowers, like daisies) follows a fixed mathematical pattern. The nth seed is at position

$$r = \sqrt{n},$$

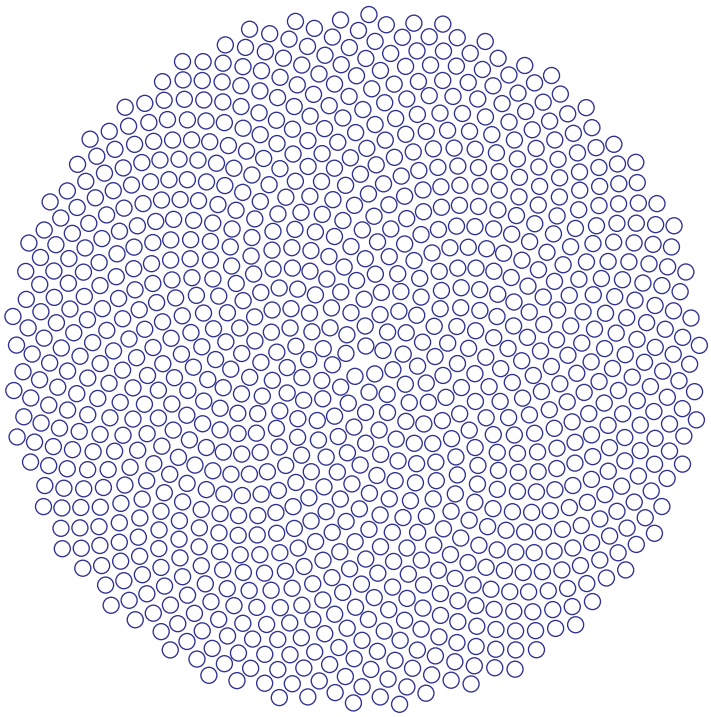


Figure 7.16 A perfect sunflower?

with angular coordinate $\pi dn/180$ radians, where d is the constant angle of divergence (in degrees) between any two successive seeds, i.e. between the nth and $(n+1)$th seeds. A perfect sunflower head (Figure 7.16) is generated by $d = 137.51°$. Write a program to plot the seeds; use a circle (○) for each seed. A remarkable feature of this model is that the angle d must be exact to get proper sunflowers. Experiment with some different values, e.g. 137.45° (spokes, from fairly far out), 137.65° (spokes all the way), 137.92° (Catherine wheels).

7.5 The equation of an ellipse in polar coordinates is given by

$$r = a(1 - e^2)/(1 - e\cos\theta),$$

where a is the semi-major axis and e is the eccentricity, if one focus is at the origin, and the semi-major axis lies on the x-axis.

Halley's Comet, which visited us in 1985/6, moves in an elliptical orbit about the Sun (at one focus) with a semi-major axis of 17.9 A.U. (A.U. stands for Astronomical Unit, which is the mean distance of the Earth from the Sun: 149.6 million km.) The eccentricity of the orbit is 0.967276. Write a program which draws the orbit of Halley's Comet and the Earth (assume the Earth is circular).

7.6 A very interesting iterative relationship that has been studied a lot recently is defined by

$$y_{k+1} = r y_k (1 - y_k)$$

(this is a discrete form of the well-known *logistic model*). Given y_0 and r, successive y_k's may be computed very easily, e.g. if $y_0 = 0.2$ and $r = 1$, then $y_1 = 0.16$, $y_2 = 0.1334$, and so on.

This formula is often used to model population growth in cases where the growth is not unlimited, but is restricted by shortage of food, living area, etc.

y_k exhibits fascinating behavior, known as *mathematical chaos*, for values of r between 3 and 4 (independent of y_0). Write a program which plots y_k against k (as individual points).

Values of r that give particularly interesting graphs are 3.3, 3.5, 3.5668, 3.575, 3.5766, 3.738, 3.8287, and many more that can be found by patient exploration.

7.7 A rather beautiful *fractal* picture can be drawn by plotting the points (x_k, y_k) generated by the following difference equations

$$x_{k+1} = y_k(1 + \sin 0.7x_k) - 1.2\sqrt{|x_k|},$$
$$y_{k+1} = 0.21 - x_k,$$

starting with $x_0 = y_0 = 0$. Write a program to draw the picture (plot individual points; do not join them).

8 Loops

The objectives of this chapter are to enable you to learn to:

- **Program (or code) determinate loops with `for`.**
- **Program (or code) indeterminate loops with `while`.**

In Chapter 2 we introduced the powerful `for` statement, which is used to repeat a block of statements a fixed number of times. This type of structure, where the number of repetitions must be determined in advance, is sometimes called *determinate repetition*. However, it often happens that the condition to end a loop is only satisfied *during the execution of the loop itself*. Such a structure is called *indeterminate*. This chapter is mainly about indeterminate loops, but to emphasize the difference between the two types of repetition we will first look at some more examples of `for` loops.

8.1 Determinate repetition with `for`

8.1.1 Binomial coefficient

The *binomial coefficient* is widely used in mathematics and statistics. It is defined as the number of ways of choosing r objects out of n without regard to order, and is given by

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}. \tag{8.1}$$

If this form is used, the factorials can get very big, causing an overflow. But a little thought reveals we can simplify Equation (8.1) as follows:

$$\binom{n}{r} = \frac{n(n-1)(n-2)\cdots(n-r+1)}{r!}, \tag{8.2}$$

$$\text{e.g.} \quad \binom{10}{3} = \frac{10!}{3! \times 7!} = \frac{10 \times 9 \times 8}{1 \times 2 \times 3}.$$

Using Equation (8.2) is computationally much more efficient:

```
ncr = 1;
n = ...
r = ...

for k = 1:r
  ncr = ncr * (n - k + 1) / k;
end

disp( ncr )
```

The binomial coefficient is sometimes pronounced 'n-see-r'. Work through the program by hand with some sample values.

8.1.2 Update processes

Many problems in science and engineering involve modeling a process where the main variable is repeatedly updated over a period of time. Here is an example of such an *update process*.

A can of orange juice at temperature 25°C is placed in a fridge, where the ambient temperature F is 10°C. We want to find out how the temperature of the orange juice changes over a period of time. A standard way of approaching this type of problem is to break the time period up into a number of small steps, each of length dt. If T_i is the temperature at the *beginning* of step i, we can use the following model to get T_{i+1} from T_i:

$$T_{i+1} = T_i - K\,dt\,(T_i - F), \tag{8.3}$$

where K is a constant parameter depending on the insulating properties of the can, and the thermal properties of orange juice. Assume that units are chosen so that time is in minutes.

The following script implements this scheme. To make the solution more general, take the starting time as a, and the end time as b. If dt is very small, it will be inconvenient to have output displayed after every step, so the script also asks you for the *output interval* `opint`. This is the time (in minutes) between successive rows of output. It checks that this interval is an integer multiple of dt. Try the script with some sample values, e.g. $dt = 0.2$ minutes and `opint` = 5 minutes (these are the values used for the output below).

```
K = 0.05;
F = 10;
a = 0;                  % start time
b = 100;                % end time
time = a;               % initialize time
T = 25;                 % initialize temperature
load train              % prepare to blow the whistle
dt = input( 'dt: ' );
opint = input( 'output interval (minutes): ' );
if opint/dt ~= fix(opint/dt)
  sound(y, Fs)          % blow the whistle!
  disp( 'output interval is not a multiple of dt!' );
  break
end

clc
format bank
disp( '          Time           Temperature' );
disp( [time T] )      % display initial values

for time = a+dt : dt : b
  T = T - K * dt * (T - F);
  if abs(rem(time, opint)) < 1e-6    % practically zero!
    disp( [time T] )
  end
end
```

Output:

```
 Time           Temperature
     0          25.00
  5.00          21.67
   ...
 95.00          10.13
100.00          10.10
```

Note:

1. The function `rem` is used to display the results every `opint` minutes: when `time` is an integer multiple of `opint` its remainder when divided by `opint` should be zero. However, due to rounding error, the remainder is not always *exactly* zero. It is therefore better to test whether its absolute value is less than some very small value. (Rounding error is discussed in Chapter 9).
2. While this is probably the most obvious way of writing the script, we cannot easily plot the graph of temperature against time this way, since `time` and

`T` are scalars which are repeatedly updated. To plot the graph they both need to be vectors (see Chapter 11).

3. Note how sound is implemented. See `help audio` for other interesting sounds supplied by MATLAB.
4. In case you are wondering how I got the headings in the right place, I'll let you into the secret. Run the script without a heading but with the numerical output as you want it. Then simply paste the `disp` statement with the headings into the command window and edit it until the headings fall in the right place. Paste the final version of the `disp` statement into the script.
5. Note the use of `break` to stop the script prematurely if the user gives bad input. See below for more general use of `break`.

8.1.3 Nested `for`s

As we saw in Chapter 6 (**Vectorizing nested `for`s**), `for` loops can be nested inside each other. The main point to note is that the index of the *inner* `for` moves faster.

8.2 Indeterminate repetition with `while`

Determinate loops all have in common the fact that you can work out in principle exactly how many repetitions are required *before the loop starts*. But in the next example, there is no way *in principle* of working out the number of repeats, so a different structure is needed.

8.2.1 A guessing game

The problem is easy to state. MATLAB 'thinks' of an integer between 1 and 10 (i.e. generates one at random). You have to guess it. If your guess is too high or too low, the script must say so. If your guess is correct, a message of congratulations must be displayed.

A little more thought is required here, so a structure plan might be helpful:

1. Generate random integer
2. Ask user for guess
3. While guess is wrong:
   If guess is too low
      Tell her it is too low
   Otherwise

Tell her it is too high
Ask user for new guess

4. Polite congratulations
5. Stop.

Here is the script:

```
matnum = floor(10 * rand + 1);
guess = input( 'Your guess please: ' );
load splat

while guess ~= matnum
sound(y, Fs)

  if guess > matnum
    disp( 'Too high' )
  else
    disp( 'Too low' )
  end;

  guess = input( 'Your next guess please: ' );
end

disp( 'At last!' )
load handel
sound(y, Fs)                % hallelujah!
```

Try it out a few times. Note that the `while` loop repeats as long as `matnum` is not equal to `guess`. There is no way in principle of knowing how many loops will be needed before the user guesses correctly. The problem is truly indeterminate.

Note that `guess` has to be input in two places: firstly to get the `while` loop going, and secondly during the execution of the `while`.

8.2.2 The `while` statement

In general the `while` statement looks like this:

```
while  condition
   statements
end
```

The `while` construct repeats *statements* WHILE its *condition* remains true. The condition therefore is the condition to repeat once again. The condition is tested each time BEFORE *statements* are repeated. Since the condition is evaluated before *statements* are executed, it is possible to arrange for *statements* not to be executed at all under certain circumstances. Clearly, *condition* must depend on *statements* in some way, otherwise the loop will never end.

Recall that a *vector* condition is considered true only if *all* its elements are non-zero.

The command-line form of `while` is:

`while` *condition statements*`, end`

8.2.3 Doubling time of an investment

Suppose we have invested some money which draws 10 percent interest per year, compounded. We would like to know how long it takes for the investment to double. More specifically, we want a statement of the account each year, *until* the balance has doubled. The English statement of the problem hints heavily that we should use an indeterminate loop with the following structure plan:

1. Initialize balance, year, interest rate
2. Display headings
3. Repeat
   Update balance according to interest rate
   Display year, balance
   until balance exceeds twice original balance
4. Stop.

A program to implement this plan would be

```
a = 1000;
r = 0.1;
bal = a;
year = 0;
disp( 'Year       Balance' )

while bal < 2 * a
  bal = bal + r * bal;
  year = year + 1;
  disp( [year bal] )
end
```

Note that the more natural phrase in the structure plan 'until balance exceeds twice original balance' must be coded as

```
while bal < 2 * a ...
```

This condition is checked each time before another loop is made. Repetition occurs only if the condition is true. Here's some output (for an opening balance of $1000):

```
Year        Balance

   1        1100.00
   2        1210.00
   3        1331.00
   4        1464.10
   5        1610.51
   6        1771.56
   7        1948.72
   8        2143.59
```

Note that when the last loop has been completed, the condition to repeat is false for the first time, since the new balance ($2143.59) is more than $2000. Note also that a determinate `for` loop *cannot* be used here because we don't know how many loops are going to be needed until after the script has run (although in this particular example perhaps you *could* work out in advance how many repeats are needed?).

If you want to write the new balance only while it is *less* than $2000, all that has to be done is to move the statement

```
disp( [year bal] )
```

until it is the first statement in the `while` loop. Note that the initial balance of $1000 is displayed now.

8.2.4 Prime numbers

Many people are obsessed with prime numbers, and most books on programming have to include an algorithm to test if a given number is prime. So here's mine.

A number is prime if it is not an exact multiple of any other number except itself and 1, i.e. if it has no factors except itself and 1. The easiest plan of attack then is as follows. Suppose *P* is the number to be tested. See if any numbers *N* can be found that divide into *P* without remainder. If there are none,

P is prime. Which numbers N should we try? Well, we can speed things up by restricting P to odd numbers, so we only have to try odd divisors N. When do we stop testing? When $N = P$? No, we can stop a lot sooner. In fact, we can stop once N reaches $\sqrt{P}$, since if there is a factor greater than $\sqrt{P}$ there must be a corresponding one less than $\sqrt{P}$, which we would have found. And where do we start? Well, since $N = 1$ will be a factor of any P, we should start at $N = 3$. The structure plan is as follows:

1. Input P
2. Initialize N to 3
3. Find remainder R when P is divided by N
4. While $R \neq 0$ and $N < \sqrt{P}$ repeat:
   Increase N by 2
   Find R when P is divided by N
5. If $R \neq 0$ then
   P is prime
   Else
   P is not prime
6. Stop.

Note that there may be no repeats—R might be zero the first time. Note also that there are *two* conditions under which the loop may stop. Consequently, an `if` is required after completion of the loop to determine which condition stopped it.

See if you can write the script. Then try it out on the following numbers: 4058879 (not prime), 193707721 (prime) and 2147483647 (prime). If such things interest you, the largest *known* prime number at the time of writing was $2^{6972593}$–1 (discovered in June 1999). It has 2098960 digits and would occupy about 70 pages if it was printed in a newspaper. Obviously our algorithm cannot test such a large number, since it's unimaginably greater than the largest number which can be represented by MATLAB. Ways of testing such huge numbers for primality are described in D.E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms* (Addison-Wesley, 1981). This particular whopper was found by the GIMPS (Great Internet Mersenne Prime Search). See http://www.utm.edu/research/primes/largest.html for more information on the largest known primes.

8.2.5 Projectile trajectory

In Chapter 3 we considered the flight of a projectile, given the usual equations of motion (assuming no air resistance). We would like to know now when and where it will hit the ground. Although this problem can be solved with a determinate loop (if you know enough applied mathematics), it is of interest also to see

how to solve it with an indeterminate `while` loop. The idea is to calculate the trajectory repeatedly with increasing time, *while* the vertical displacement (y) remains positive. Here's the script:

```
dt = 0.1;
g = 9.8;
u = 60;
ang = input( 'Launch angle in degrees: ' );
ang = ang * pi / 180;            % convert to radians
x = 0; y = 0; t = 0;             % for starters
more(15)

while y >= 0
  disp( [t x y] );
  t = t + dt;
  y = u * sin(ang) * t - g * t^2 / 2;
  x = u * cos(ang) * t;
end
```

The command `more(n)` gives you `n` lines of output before pausing. This is called *paging*. To get another single line of output press **Enter**. To get the next page of `n` lines press the spacebar. To quit the script press **q**.

Try the script for some different launch angles. Can you find the launch angle which gives the maximum horizontal range (x)? What launch angle keeps it in the air for the longest time?

Note that when the loop finally ends, the value of `y` will be negative (check this by displaying `y`). However, the position of the `disp` statement ensures that only positive values of `y` are displayed. If for some reason you need to record the last value of `t`, say, before `y` becomes negative, you will need an `if` statement inside the `while`, e.g.

```
if y >= 0
  tmax = t;
end
```

Change the script so that it displays the last time for which `y` was positive (`tmax`), *after* the `while` loop has ended.

Now suppose we want to plot the trajectory, as shown in Figure 8.1. Note in particular how the trajectory stops above the x-axis. We need to use vectors now. Here is the script:

```
dt = 0.1;
g = 9.8;
u = 60;
```

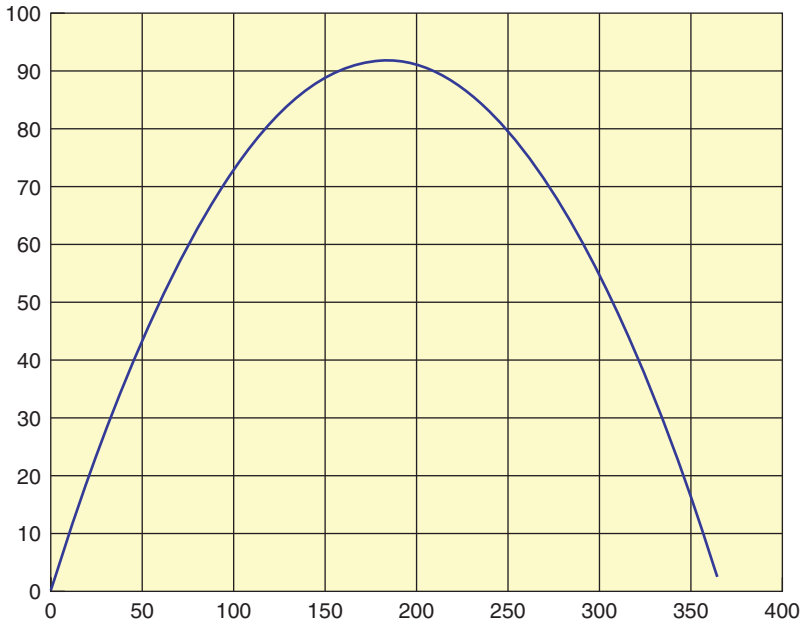


Figure 8.1 Projectile trajectory

```
ang = input( 'Launch angle in degrees: ' );
ang = ang * pi / 180;              % convert to radians
xp = zeros(1); yp = zeros(1);      % initialize
y = 0; t = 0;
i = 1;                % initial vector subscript

while y >= 0
  t = t + dt;
  i = i + 1;
  y = u * sin(ang) * t - g * t^2 / 2;
  if y >= 0
    xp(i) = u * cos(ang) * t;
    yp(i) = y;
  end
end

plot(xp, yp),grid
```

Note that the function `zeros` is used to initialize the vectors. This also clears any vector of the same name hanging around in the workspace from previous runs.

Note also the use of an `if` inside the `while` loop to ensure that only coordinates of points above the ground are added to the vectors `xp` and `yp`.

If you want the last point above the ground to be closer to the ground, try a smaller value of `dt`, e.g. 0.01.

8.2.6 `break` and `continue`

Any loop structure you are likely to encounter in scientific programming can be coded with either 'pure' `for` or `while` loops, as illustrated by the examples in this chapter. However, as a concession to intellectual laziness I feel obliged to mention in passing the `break` and `continue` statements.

If there are a number of different conditions to stop a `while` loop you may be tempted to use a `for` with the number of repetitions set to some accepted cut-off value (or even `Inf`) but enclosing `if` statements which `break` out of the `for` when the various conditions are met. Why is this not regarded as the best programming style? The reason is simply that when you read the code months later you will have to wade through the whole loop to find all the conditions to end it, rather than see them all paraded at the start of the loop in the `while` clause.

If you are going to insist on using `break` you will have to look it up in `help` for yourself!

The `continue` statement is somewhat less virulent than `break` ...

8.2.7 Menus

Try the following program, which sets up a menu window, as in Figure 8.2:

```
k = 0;

while k ~= 3
  k = menu( 'Click on your option', 'Do this', ...
            'Do that', 'Quit' );
  if k == 1
    disp( 'Do this ... press any key to continue ...' )
    pause
  elseif k == 2
    disp( 'Do that ... press any key to continue ...' )
    pause
  end

end;
```

Figure 8.2 A menu window

Note:

1. The `menu` function enables you to set up a menu of choices for a user.
2. `menu` takes only string arguments. The first one is the title of the menu. The second and subsequent strings are the choices available to the user.
3. The value returned by `menu` (`k` here) numbers the user's choices.
4. Since one has no idea how many choices the user will make, `menu` is properly enclosed in an indeterminate `while` loop. The loop continues to present the menu until the last option (in this example) is selected.
5. You can design much more sophisticated menu-driven applications with the MATLAB GUIDE (Graphical User Interface Development Environment).

Summary

- A `for` statement should be used to program a determinate loop, where the number of repeats is known (in principle) *before* the loop is encountered. This situation is characterized by the general structure plan:

  Repeat *N* times:
  Block of statements to be repeated

  where *N* is known or computed *before* the loop is encountered for the first time, and is not changed by the block.

- A `while` statement should be used to program an indeterminate repeat structure, where the exact number of repeats is *not* known in advance. Another way of saying this is that these statements should be used whenever the truth value of the

condition for repeating is changed in the body of the loop. This situation is characterized by the following structure plan:

While *condition* is true repeat:
    statements to be repeated (reset truth value of *condition*).

Note that *condition* is the condition to repeat.

- The statements in a `while` construct may sometimes never be executed.
- Loops may be nested to any depth.
- The `menu` statement inside a `while` loop may be used to present a user with a menu of choices.

EXERCISES

8.1 A person deposits $1000 in a bank. Interest is compounded monthly at the rate of 1 percent per month. Write a program which will compute the monthly balance, but write it only *annually* for 10 years (use nested `for` loops, with the outer loop for 10 years, and the inner loop for 12 months). Note that after 10 years, the balance is $3300.39, whereas if interest had been compounded annually at the rate of 12 percent per year the balance would only have been $3105.85.

See if you can vectorize your solution.

8.2 There are many formulae for computing π (the ratio of a circle's circumference to its diameter).
The simplest is

$$\frac{\pi}{4} = 1 - 1/3 + 1/5 - 1/7 + 1/9 - \cdots \tag{8.4}$$

which comes from putting $x = 1$ in the series

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots \tag{8.5}$$

(a) Write a program to compute π using Equation (8.4). Use as many terms in the series as your computer will reasonably allow (start modestly, with 100 terms, say, and rerun your program with more and more each time). You should find that the series converges very slowly, i.e. it takes a lot of terms to get fairly close to π.

(b) Rearranging the series speeds up the convergence:

$$\frac{\pi}{8} = \frac{1}{1 \times 3} + \frac{1}{5 \times 7} + \frac{1}{9 \times 11} \cdots$$

Write a program to compute π using this series instead. You should find that you need fewer terms to reach the same level of accuracy that you got in (a).

(c) One of the fastest series for π is

$$\frac{\pi}{4} = 6\arctan\frac{1}{8} + 2\arctan\frac{1}{57} + \arctan\frac{1}{239}.$$

Use this formula to compute π. Don't use the MATLAB function `atan` to compute the arctangents, since that would be cheating. Rather use Equation (8.5).

(d) Can you vectorize any of your solutions (if you haven't already)?

8.3 The following method of computing π is due to Archimedes:

1. Let $A = 1$ and $N = 6$
2. Repeat 10 times, say:
   Replace N by $2N$
   Replace A by $[2 - \sqrt{(4 - A^2)}]^{1/2}$
   Let $L = NA/2$
   Let $U = L/\sqrt{1 - A^2/2}$
   Let $P = (U + L)/2$ (estimate of π)
   Let $E = (U - L)/2$ (estimate of error)
   Print N, P, E
3. Stop.

Write a program to implement the algorithm.

8.4 Write a program to compute a table of the function

$$f(x) = x\sin\left[\frac{\pi(1 + 20x)}{2}\right]$$

over the (closed) interval $[-1, 1]$ using increments in x of (a) 0.2 (b) 0.1 and (c) 0.01.

Use your tables to sketch graphs of $f(x)$ for the three cases (by hand), and observe that the tables for (a) and (b) give totally the wrong picture of $f(x)$.

Get your program to draw the graph of $f(x)$ for the three cases, superimposed.

8.5 The transcendental number e (2.71828182845904 ...) can be shown to be the limit of

$$(1 + x)^{1/x}$$

as x tends to zero (from above). Write a program which shows how this expression converges to e as x gets closer and closer to zero.

8.6 A square wave of period T may be defined by the function

$$f(t) = \begin{cases} 1 & (0 < t < T) \\ -1 & (-T < t < 0). \end{cases}$$

The Fourier series for $f(t)$ is given by

$$F(t) = \frac{4}{\pi} \sum_{k=0}^{\infty} \frac{1}{2k+1} \sin\left[\frac{(2k+1)\pi t}{T}\right].$$

It is of interest to know how many terms are needed for a good approximation to this infinite sum. Taking $T = 1$, write a program to compute and plot the sum to n terms of the series for t from -1.1 to 1.1 in steps of 0.01, say. Run the program for different values of n, e.g. 1, 3, 6, etc.

Superimpose plots of $F(t)$ against t for a few values of n.

On each side of a discontinuity a Fourier series exhibits peculiar oscillatory behavior known as the Gibbs phenomenon. Figure 8.3 shows this clearly for the above series with $n = 20$ (and increments in t of 0.01). The phenomenon is much sharper for $n = 200$ and t increments of 0.001.

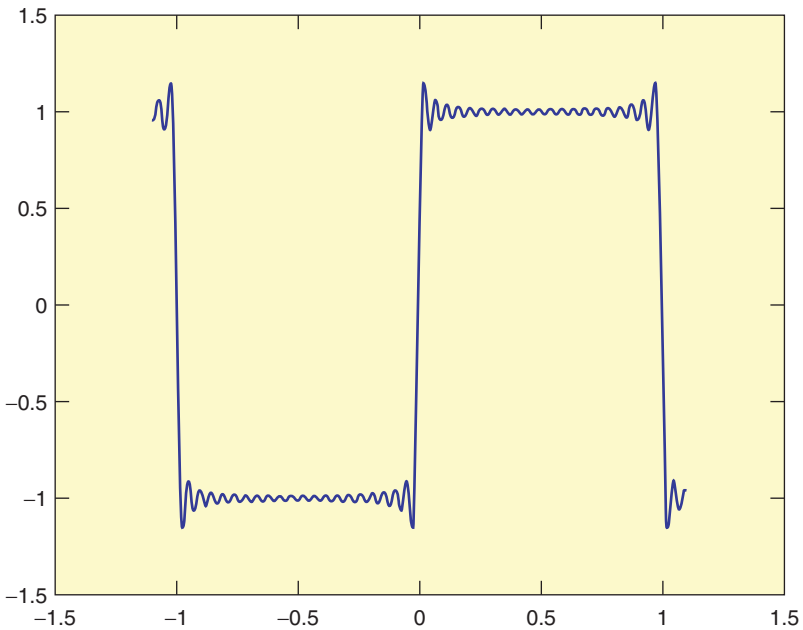


Figure 8.3 Fourier series: Gibbs phenomenon

8.7 If an amount of money A is invested for k years at a nominal annual interest rate r (expressed as a decimal fraction), the value V of the investment after k years is given by

$$V = A(1 + r/n)^{nk}$$

where n is the number of compounding periods per year. Write a program to compute V as n gets larger and larger, i.e. as the compounding periods become more and more frequent, like monthly, daily, hourly, etc. Take $A = 1000$, $r = 4$ percent and $k = 10$ years. You should observe that your output gradually approaches a limit. **Hint**: Use a `for` loop which doubles n each time, starting with $n = 1$.

Also compute the value of the formula Ae^{rk} for the same values of A, r and k (use the MATLAB function `exp`), and compare this value with the values of V computed above. What do you conclude?

8.8 Write a program to compute the sum of the series $1^2 + 2^2 + 3^2 \ldots$ such that the sum is as large as possible without exceeding 1000. The program should display how many terms are used in the sum.

8.9 One of the programs in Section 8.2 shows that an amount of $1000 will double in eight years with an interest rate of 10 percent. Using the same interest rate, run the program with initial balances of $500, $2000 and $10 000 (say) to see how long they all take to double. The results may surprise you.

8.10 Write a program to implement the structure plan of Exercise 3.2.

8.11 Use the Taylor series

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

to write a program to compute $\cos x$ correct to four decimal places (x is in radians). See how many terms are needed to get 4-figure agreement with the MATLAB function `cos`. Don't make x too large; that could cause rounding error.

8.12 A student borrows $10 000 to buy a used car. Interest on her loan is compounded at the rate of 2 percent per month while the outstanding balance of the loan is more than $5000, and at 1 percent per month otherwise. She pays back $300 every month, except for the last month, when the repayment must be less than $300. She pays at the end of the month, *after* the interest on the balance has been compounded. The first repayment is made one month after the loan is paid out. Write a program which displays a monthly statement of the balance (after the monthly payment has been made), the final payment, and the month of the final payment.

8.13 A projectile, the equations of motion of which are given in Chapter 4, is launched from the point O with an initial velocity of 60 m/s at an angle of 50°

to the horizontal. Write a program which computes and displays the time in the air, and horizontal and vertical displacement from the point O every 0.5 seconds, as long as the projectile remains above a horizontal plane through O.

8.14 When a resistor (R), capacitor (C) and battery (V) are connected in series, a charge Q builds up on the capacitor according to the formula

$$Q(t) = CV(1 - e^{-t/RC})$$

if there is no charge on the capacitor at time $t = 0$. The problem is to monitor the charge on the capacitor every 0.1 seconds in order to detect when it reaches a level of 8 units of charge, given that $V = 9$, $R = 4$ and $C = 1$. Write a program which displays the time and charge every 0.1 seconds until the charge first exceeds 8 units (i.e. the last charge displayed must exceed 8). Once you have done this, rewrite the program to display the charge only while it is strictly less than 8 units.

8.15 Adapt your program for the prime number algorithm in Section 8.2 to find all the prime factors of a given number (even or odd).

9

Errors and pitfalls

The objective of this chapter is to enable you to:

- **Begin to recognize and avoid different sorts of errors and pitfalls.**

Even experienced programmers seldom get programs to run correctly the first time. In computer jargon, an error in a program is called a *bug*. The story is that a hapless moth short-circuited two thermionic valves in one of the earliest computers. This primeval (charcoaled) 'bug' took days to find. The process of detecting and correcting such errors is therefore called *debugging*. There are a number of different types of errors and pitfalls, some of which are peculiar to MATLAB, and some of which may occur when programming in any language. These are discussed briefly in this chapter.

9.1 Syntax errors

Syntax errors are typing errors in MATLAB statements (e.g. `plog` instead of `plot`). They are the most frequent type of error, and are *fatal*: MATLAB stops execution and displays an error message. Since MATLAB is not even remotely as intelligent as you are, the messages can sometimes be rather unhelpful—even misleading. A few common examples are given below.

```
")" expected, "end of line" found.
```

This is generated in response to something like

```
2*(1+3
```

What could possibly be more helpful than that?

You also get this message, quite properly, after something like

```
disp(['the answer is ' num2str(2)]
```

because the closing right parenthesis has been omitted. However, if you omit the closing square bracket instead, you get the somewhat unhelpful `";" expected, ")" found.`

```
"end" expected, "End of Input" found.
```

This seems pretty obvious, and usually is. But you also get it after the following

```
x = input( 'x: ' );
if x < 0
  disp( 'negative' )
else if x == 0
  disp( 'zero' )
else
  disp( 'positive' )
end
```

where `end` is *not* missing. The actual error is that `else if` has been typed as *two* words instead of one, as required in an `elseif` ladder. The effect of this is to open a new (nested) `if` statement as part of the `else`. This naturally requires its own `end`. In fact, adding an extra `end` at the end will suppress the error message, but may not give you what you expected (see **Nested** `if`s in Chapter 2).

```
Error using ==> *   Inner matrix dimensions must agree.
```

This is generated for example by

```
[1 2 3] * [4 5 6]
```

since *matrix* multiplication cannot be applied to two row vectors. Presumably the array operation `.*` was intended. You get a similar message after something like

```
[1 2 3] + [4 5]
```

because the vectors don't have the same number of elements.

```
Missing operator, comma, or semicolon.
```

This is a very common message. It is generated, for example, by the following statement:

```
if x < 0 disp('negative') end
```

(as long as `x` is defined) because there is no comma after the `disp` statement, or after

```
2(1+3)
```

because `*` (say) was left out after the 2.

But it can be misleading. For example, it occurs after the statement

```
2x = 3
```

(perhaps you meant to type `x2` for a variable name). Maybe MATLAB thought you were trying to evaluate a logical expression like `2*x == 3` in which case two operators were omitted. Who knows?

You also get it after the following statement, which a BASIC programmer might try:

```
name$ = 'Joe'
```

```
Expected a variable, function, or constant, found ...
```

This message occurs, quite naturally, after something like

```
x =
```

(you simply forgot to complete the statement) or

```
y = sin()
```

(you left out the argument of `sin`).

But you get the misleading variation `Missing variable or function` with

```
disp( "Hi there" )
```

or

```
king = "Henry VIII"
```

(for using double instead of single quotes).

There are a large number of possible syntax errors—you will probably have discovered a good few yourself. With experience you will gradually become more adept at spotting your mistakes.

9.1.1 `lasterr`

The function `lasterr` returns the last error message generated.

9.2 Pitfalls and surprises

9.2.1 Incompatible vector sizes

Consider the following statements:

```
x = 0:pi/20:3*pi;
y = sin(x);
x = 0:pi/40:3*pi;
plot(x,y)
```

You'll get the error message

```
Error using ==> plot
Vectors must be the same lengths.
```

because you forgot to recalculate `y` after reducing the `x` increments. `whos` reveals the problem:

```
x          1x121          ...
y          1x61           ...
```

9.2.2 Name hiding

Recall that a workspace variable 'hides' a script or function of the same name. The only way to access such a script or function is to clear the offending variable from the workspace.

Furthermore, a MATLAB function hides a script of the same name, e.g. create a script called `why.m` that displays some junk message, and then type `why` at the command line.

If you are worried that a variable or script which you are thinking of creating, say `blob`, may be a MATLAB function, try `help blob` first.

9.2.3 Other pitfalls for the unwary

It may be a good idea to work through the following sections in Chapter 2 again: **Vectorization of formulae: Danger!** and **A common mistake: `for`less loops!**.

You might wonder why MATLAB is used at all, since it appears to have so many ways of catching you out. There are two responses to this. Firstly, other languages have different pitfalls. Secondly, the fact that powerful motorcars can kill people has never been accepted as an excuse for not owning or driving one. You simply have to learn the rules of the road and drive carefully to avoid accidents.

9.3 Errors in logic

These are errors in the actual algorithm you are using to solve a problem, and are the most difficult to find; the program runs, but gives the wrong answers! It's even worse if you don't realize the answers are wrong. The following tips might help you to check out the logic.

- Try to run the program for some special cases where you know the answers.
- If you don't know any exact answers, try to use your insight into the problem to check whether the answers seem to be of the right order of magnitude.
- Try working through the program by hand (or use MATLAB's excellent interactive debugging facilities—see Chapter 10) to see if you can spot where things start going wrong.

9.4 Rounding error

At times, as we have seen, a program will give numerical answers which we know are wrong. This can also be due to *rounding error*, which results from the finite precision available on the computer, i.e. eight bytes per variable, instead of an infinite number.

Run the following program:

```
x = 0.1;
while x ~= 0.2
  x = x + 0.001;
  fprintf( '%g  %g\n', x, x - 0.2 )
end
```

You will find that you need to crash the program to stop it, i.e. with **Ctrl-break** on a PC. The variable `x` never has the value 0.2 *exactly*, because of rounding error. In fact, `x` misses 0.2 by about 8.3×10^{-17}, as can be seen from displaying the value of `x - 0.2`. It would be better to replace the `while` clause with

```
while x <= 0.2
```

or, even better, with

```
while abs(x - 0.2) > 1e-6
```

In general, it is always better to test for 'equality' of two non-integer expressions as follows:

```
if abs((a-b)/a) < 1e-6 disp( 'a practically equals b' ),
end
```

or

```
if abs((a-b)/b) < 1e-6 ...
```

Note that this equality test is based on the *relative* difference between `a` and `b`, rather than on the *absolute* difference.

Rounding error may sometimes be reduced by a mathematical rearrangement of a formula. Recall yet again the common or garden quadratic equation

$$ax^2 + bx + c = 0,$$

with solutions

$$x_1 = (-b + \sqrt{b^2 - 4ac})/(2a),$$
$$x_2 = (-b - \sqrt{b^2 - 4ac})/(2a).$$

Taking $a = 1$, $b = -10^7$ and $c = 0.001$ gives $x_1 = 10^7$ and $x_2 = 0$. The second root is expressed as the difference between two nearly equal numbers, and considerable significance is lost. However, as you no doubt remember, the product of the two roots is given by c/a. The second root can therefore be expressed as $(c/a)/x_1$. Using this form gives $x_2 = 10^{-10}$, which is more accurate.

9.5 Trapping and generating errors

The functions `eval` and `lasterr` may be used to trap errors when a user is trying to input a valid MATLAB expression. See Chapter 12 for an example involving these functions in a `try/catch` structure, which may be used to recover gracefully from an error (i.e. without the script crashing).

The `error` statement displays an error message and returns control to the keyboard. For example, a program to compute the roots of the quadratic equation given above could have the following code segment to prevent division by zero:

```
if (a == 0)
   error( 'if a = 0 equation is not a quadratic,
           dum-dum!' )
end
```

Summary

- Syntax errors are mistakes in the construction of MATLAB statements.
- Logical errors are errors in the algorithm used to solve a problem.
- Rounding error occurs because a computer can store numbers only to a finite accuracy.

EXERCISES

9.1 The Newton quotient

$$\frac{f(x+h) - f(x)}{h}$$

may be used to estimate the first derivative $f'(x)$ of a function $f(x)$, if h is 'small'. Write a program to compute the Newton quotient for the function

$$f(x) = x^2$$

at the point $x = 2$ (the exact answer is 4) for values of h starting at 1, and decreasing by a factor of 10 each time (use a `for` loop). The effect of rounding error becomes apparent when h gets 'too small', i.e. less than about 10^{-12}.

9.2 The solution of the set of simultaneous equations

$$ax + by = c$$
$$dx + ey = f$$

(Exercise 3.6) is given by

$$x = (ce - bf)/(ae - bd),$$
$$y = (af - cd)/(ae - bd).$$

If $(ae - bd)$ is small, rounding error may cause quite large inaccuracies in the solution. Consider the system

$$0.2038x + 0.1218y = 0.2014,$$
$$0.4071x + 0.2436y = 0.4038.$$

Show that with four-figure floating point arithmetic the solution obtained is $x = -1$, $y = 3$. This level of accuracy may be simulated in the solution of Exercise 3.6 with some statements like

```
ae = floor( a * e * 1e4 ) / 1e4
```

and appropriate changes in the coding. The exact solution, obtained without rounding, is $x = -2$, $y = 5$. If the coefficients in the equations are themselves subject to experimental error, the 'solution' of this system using limited accuracy is totally meaningless.

MATLAB can of course solve linear equations directly, as we shall see in Chapter 16.

10

Function M-files

The objective of this chapter is to enable you to:

- **Represent your own functions with inline objects and function M-files.**

We have already seen that MATLAB has a number of *built-in* (compiled) functions, e.g. `sin`, `sqrt`, `sum`. You can verify that they are built-in by attempting to `type` them. Try `type sin` for example. MATLAB also has some functions in the form of *function M-files*, e.g. `fzero`, `why`. You can see what's in them with `type`, e.g. `type why`.

MATLAB enables you to create your own function M-files. A function M-file is similar to a script file in that it also has a `.m` extension. However, a function M-file differs from a script file in that a function M-file communicates with the MATLAB workspace only through specially designated *input* and *output arguments*.

Functions are indispensable when it comes to breaking a problem down into manageable logical pieces.

Short mathematical functions may be written as one-line *inline objects*.

This chapter outlines how to write your own functions, and also introduces MATLAB's interactive debugging facilities.

10.1 Some examples

10.1.1 Inline objects: harmonic oscillators

If two coupled harmonic oscillators, e.g. two masses connected with a spring on a very smooth table, are considered as a single system, the output of the

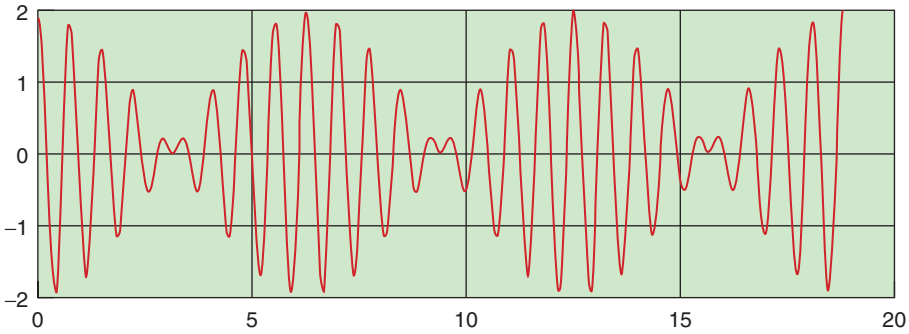


Figure 10.1 $\cos(8t) + \cos(9t)$

system as a function of time t could be given by something like

$$h(t) = \cos(8t) + \cos(9t). \tag{10.1}$$

You can represent $h(t)$ at the command line by creating an *inline object* as follows:

```
h = inline( 'cos(8*t) + cos(9*t)' );
```

Now write some MATLAB statements in the Command Window which use your function `h` to draw the graph in Figure 10.1, e.g.

```
x = 0 : pi/40 : 6*pi;
plot(x, h(x)), grid
```

Note:

- The variable `t` in the `inline` definition of `h` is the *input argument*. It is essentially a 'dummy' variable, and serves only to provide input to the function from the outside world. You can use any variable name here; it doesn't have to be the same as the one used when you invoke (use) the function.
- You can create functions of more than one argument with `inline`. For example,

```
f = inline( 'x.^2 + y.^2', 'x', 'y' );
f(1, 2)
ans =
     5
```

10.1.2 Function M-files: Newton's method again

Newton's method may be used to solve a general equation $f(x) = 0$ by repeating the assignment

$$x \text{ becomes } x - \frac{f(x)}{f'(x)},$$

where $f'(x)$ (i.e. df/dx) is the first derivative of $f(x)$ (see Chapter 17 for a derivation). The process continues until successive approximations to x are close enough.

Suppose that $f(x) = x^3 + x - 3$, i.e. we want to solve the equation $x^3 + x - 3 = 0$ (another way of stating the problem is to say we want to find the *zero* of $f(x)$). We have to be able to differentiate $f(x)$. That's quite easy here: $f'(x) = 3x^2 + 1$. We could write inline objects for both $f(x)$ and $f'(x)$, but for this example we will use function M-files instead.

Use the Editor to create and save (in the current MATLAB directory) the function file `f.m` as follows:

```
function y = f(x)
y = x^3 + x - 3;
```

Then create and save another function file `df.m`:

```
function y = df(x)
y = 3*x^2 + 1;
```

Now write a *separate* script file, `newtgen.m` (in the same directory), which will stop either when the *relative* error in x is less than 10^{-8}, or after 20 steps, say:

```
% Newton's method in general
% exclude zero roots!

steps = 0;                          % iteration counter
x = input( 'Initial guess: ' ); % estimate of root
re = 1e-8;                          % required relative error
myrel = 1;

while myrel > re & (steps < 20)
  xold = x;
  x = x - f(x)/df(x);
  steps = steps + 1;
```

```
  disp( [x f(x)] )
  myrel = abs((x-xold)/x);
end;

if myrel <= re
  disp( 'Zero found at' )
  disp( x )
else
  disp( 'Zero NOT found' )
end;
```

Note that there are *two* conditions that will stop the `while` loop: convergence, or the completion of 20 steps. Otherwise the script could run indefinitely.

Here is a sample run (with `format long e`), starting with $x = 1$:

```
Initial guess: 1
    1.250000000000000e+000    2.031250000000000e-001
    1.214285714285714e+000    4.737609329445558e-003
    1.213412175782825e+000    2.779086666571118e-006
    1.213411662762407e+000    9.583445148564351e-013
    1.213411662762230e+000   -4.440892098500626e-016

Zero found at
    1.213411662762230e+000
```

Note:

- The variable `y` in the function files `f.m` and `df.m` is the *output argument*. It is a dummy variable, and defines how output will be sent back to the outside world.

By the way, you realize of course that you can use your own functions from the command line, don't you? For example,

```
≫ f(2)
```

should return 7 with `f.m` defined as above.

10.2 Basic rules

Try the following more general example, which returns the mean (`avg`) and standard deviation (`stdev`) of the values in the vector `x`. Although there are

two MATLAB functions to do this (`mean` and `std`), it is useful to have them combined into one. Write a function file `stats.m`:

```
function [avg, stdev] = stats( x )  % function definition
                                      line
% STATS         Mean and standard    % H1 line
                deviation
%               Returns mean (avg)   % Help text
                and standard
%               deviation (stdev) of
                the data in the
%               vector x, using Matlab
                functions

avg = mean(x);                       % function body
stdev = std(x);
```

Now test it in the Command Window with some random numbers, e.g.

```
r = rand(100,1);
[a, s] = stats(r);
```

Note the following points about function M-files in general:

- **General form of a function**

  A function M-file *name*`.m` has the following general form:

  ```
  function [ outarg1, outarg2, ... ] = name( inarg1, inarg2, ... )
  %  comments to be displayed with help
  ...
  outarg1 = ... ;

  outarg2 = ... ;
  ...
  ```

- `function` **keyword**

  The function file *must* start with the keyword `function` (in the function definition line).

- **Input and output arguments**

  The input and output arguments (*inarg1*, *outarg1*, etc.) are 'dummy' variables, and serve only to define the function's means of communication with the workspace. Other variable names may therefore be used in their place when the function is called (referenced).

You can think of the *actual* input arguments being copied into the dummy input arguments when the function is called. So when `stats(r)` is called in the above example, the actual input argument `r` is copied into the input argument `x` in the function file. When the function *returns* (i.e. execution of it is completed) the dummy output arguments `avg` and `std` in the function file are copied into the actual output arguments `a` and `s`.

➤ **Multiple output arguments**

If there is more than one output argument, the output arguments *must* be separated by commas and enclosed in square brackets in the function definition line, as shown.

However, when a function is called with more than one output argument, the actual output arguments may be separated by commas or spaces.

If there is only one output argument square brackets are not necessary.

➤ **Naming convention for functions**

Function names must follow the MATLAB rules for variable names.

If the filename and the function definition line name are different, the internal name is ignored.

➤ **Help text**

When you type `help` *function_name*, MATLAB displays the comment lines that appear between the function definition line and the first non-comment (executable or blank) line.

The first comment line is called the *H1 line*. The `lookfor` function searches on and displays only the H1 line. The H1 lines of all M-files in a directory are displayed under the **Description** column of the Desktop Current Directory browser.

You can make help text for an entire directory by creating a file with the name `Contents.m` that resides in the directory. This file must contain only comment lines. The contents of `Contents.m` is displayed with the command

`help` *directory_name*

If a directory does not have a `Contents.m` file, this command displays the H1 line of each M-file in the directory.

➤ **Local variables: scope**

Any variables defined inside a function are inaccessible outside the function. Such variables are called *local variables*—they exist only inside the

function, which has its own workspace separate from the base workspace of variables defined in the Command Window.

This means that if you use a variable as a loop index, say, inside a function, it will not clash with a variable of the same name in the workspace or in another function.

You can think of the *scope* of a variable as the range of lines over which the variable is accessible.

➤ **Global variables**

Variables which are defined in the base workspace are not normally accessible inside functions, i.e. their scope is restricted to the workspace itself, unless they have been declared `global`, e.g.

```
global PLINK PLONK
```

If several functions, and possibly the base workspace, all declare particular variables as global, then they all share single copies of those variables. MATLAB recommends that global variables be typed in capital letters, to remind you that they are global.

The function `isglobal(A)` returns 1 if `A` is global, and 0 otherwise.

The command `who global` gives a list of global variables.

Use `clear global` to make all variables non-global, or `clear PLINK` to make `PLINK` non-global.

➤ **Persistent variables**

A variable in a function may be declared `persistent`. Local variables normally cease to exist when a function returns. Persistent variables, however, remain in existence between function calls. A persistent variable is initialized to the empty array.

In the following example, the persistent variable `count` is used to count how many times the function `test` is called:

```
function test
persistent count
if isempty(count)
    count = 1
else
    count = count + 1
end
```

Persistent variables remain in memory until the M-file is cleared or changed, e.g.

```
clear test
```

The function `mlock` inside an M-file prevents the M-file from being cleared. A locked M-file is unlocked with `munlock`. The function `mislocked` indicates whether an M-file can be cleared or not.

The Help entry on `persistent` declares confidently: 'It is an error to declare a variable persistent if a variable with the same name exists in the current workspace.' However, this is definitely not the case at the time of writing (I tried it!).

➤ **Functions that do not return values**

You might want to write a function that doesn't return values (such functions are called *procedures* or *subroutines* in languages like Pascal and Fortran, and *void* in C++ and Java). In that case you simply omit the output argument(s) and the equal sign in the function definition line. For example, the following function will display `n` asterisks:

```
function stars(n)
asteriks = char(abs('*')*ones(1,n));
disp( asteriks )
```

Go back to Chapter 6 for an explanation of how it works.

Write such a function file (`stars.m`), and test it out, e.g. `stars(13)` should produce 13 asterisks.

➤ **Vector arguments**

It should come as no surprise that input and output arguments may be vectors, e.g. the following function generates a vector of *n* random rolls of a die:

```
function d = dice( n )
d = floor( 6 * rand(1, n) + 1 );
```

When an output argument is a vector, it is initialized each time the function is called, any previous elements being cleared. Its size at any moment is therefore determined by the most recent call of the function. For example, suppose the function `test.m` is defined as

```
function a = test
a(3) = 92;
```

Then if `b` is defined in the base workspace as

```
b =
     1     2     3     4     5     6
```

the statement

```
b = test
```

results in

```
b =
     0     0    92
```

How function arguments are passed

If a function changes the value of any of its input arguments, the change is *not reflected* in the actual input argument on return to the workspace (unless the function is called with the same input and output argument—see below). For the technically minded, input arguments appear to be passed *by value*.

You might think that passing a large matrix as an input argument by value is wasteful of memory, and you would be correct. However, the designers of MATLAB were aware of this, and so an input argument is only passed by value if a function modifies it (although the modification is not reflected on return). If a function does not modify an input argument it is passed by reference.

Simulated pass by reference

A function may be called with the same actual input and output argument. For example, the following function `prune.m` removes all the zero elements from its input argument:

```
function y = prune(x)
y = x(x ~= 0);
```

(if you can't figure out why, refer to **Subscripting with logical vectors** in Chapter 5).

You can use it to remove all the zero elements of the vector `x` as follows:

```
x = prune(x)
```

Checking the number of function arguments

A function may be called with all, some, or none of its input arguments. If called with no arguments, the parentheses must be omitted. You may not use more input arguments than appear in its definition.

The same applies to output arguments—you may specify all, some, or none of them when you use the function. If you call a function with no output arguments, the value of the first one in the definition is returned.

There are times when a function may need to know how many input/ output arguments are used on a particular call. In that case, the functions `nargin`

and `nargout` can be used to determine the number of actual input and output arguments. For example,

```
function y = foo(a, b, c);
disp( nargin );
...
```

will display the number of input arguments present on each call of `foo`.

➤ **Passing a variable number of arguments**

The functions `varargin` and `varargout` allow you to call a function with any number of input or output arguments. Since this facility involves packing arguments into a *cell array*, discussion of it is deferred to Chapter 11.

10.2.1 Subfunctions

A function M-file may contain the code for more than one function. The first function in a file is the *primary* function, and is the one invoked with the M-file name. Additional functions in the file are called *subfunctions*, and are visible only to the primary function and to other subfunctions.

Each subfunction begins with its own function definition line. Subfunctions follow each other in any order *after* the primary function.

10.2.2 Private functions

A private function is a function residing in a subdirectory with the name `private`. Private functions are visible only to functions in the parent directory. See Help for more details.

10.2.3 P-code files

The first time a function is called during a MATLAB session it is parsed ('compiled') into pseudocode and stored in memory to obviate the need for it to be parsed again during the current session. The pseudocode remains in memory until you clear it with `clear` *function_name* (see Help for all the possibilities with `clear`).

You can use the `pcode` function to save the parsed version of an M-file for use in later MATLAB sessions, or by users from whom you want to hide your

algorithms. For example, the command

```
pcode stats
```

parses `stats.m` and stores the resulting pseudocode in the file named `stats.p`.

MATLAB is very efficient at parsing so producing your own P-code files seldom makes much of a speed difference, except in the case of large GUI applications where many M-files have to be parsed before the application surfaces.

10.2.4 Improving M-file performance with the profiler

The MATLAB Profiler enables you to see where the bottlenecks in your programs are, e.g. which functions are consuming most of the time. With this information you can often redesign programs to be more efficient. See **Using MATLAB: Development Environment: Improving M-File Performance – the Profiler**.

10.3 Function handles

Our script file `newtgen.m` in Section 10.1 solves the equation $f(x) = 0$, where $f(x)$ is defined in the function file with the specific name `f.m`. This is restrictive because to solve a different equation `f.m` has to be edited first. To make `newtgen` even more general, it can be rewritten as a function M-file itself, with a *handle* to `f.m` as an input argument. This process is made possible by the built-in function `feval`, and the concept of a function handle, which we now examine.

Try the following on the command line:

```
fhandle = @sqrt;
feval(fhandle, 9)
feval(fhandle, 25)
```

Can you see that `feval(fhandle, x)` is the same as `sqrt(x)`? The statement

```
fhandle = @sqrt
```

creates a *handle* to the function `sqrt`. The handle provides a way of referring to the function, for example in a list of input arguments to another

function. A MATLAB function handle is similar to a pointer in C++, although more general.

If you have still got a function file `f.m` defined for $f(x) = x^3 + x - 3$, verify that

```
feval(@f, 2)
```

for example, returns the same value as `f(2)`.

➤ In general, the first argument of `feval` is a handle to the function to be evaluated in terms of the subsequent arguments of `feval`.

You can use `feval` inside a function to evaluate another function whose handle is passed as an argument. as we will see now. As an example, we would like to rewrite our `newtgen` script as a function `newtfun`, to be called as follows,

```
[x f conv] = newtfun( fh, dfh, x0 )
```

where `fh` and `dfh` are handles for the M-files containing $f(x)$ and $f'(x)$ respectively, and `x0` is the initial guess. The outputs are the zero, the function value at the zero, and an argument `conv` to indicate whether or not the process has converged. The complete M-file `newtfun.m` is as follows:

```
function [x, f, conv] = newtfun(fh, dfh, x0)
% NEWTON    Uses Newton's method to solve f(x) = 0.
%           fh is handle to f(x), dfh is handle to f'(x).
%           Initial guess is x0.
%           Returns final value of x, f(x), and
%           conv (1 = convergence, 0 = divergence)

steps = 0;                          % iteration counter
x = x0;
re = 1e-8;                          % required relative
                                      error
myrel = 1;

while myrel > re & (steps < 20)
    xold = x;
    x = x - feval(fh, x)/feval(dfh, x);
    steps = steps + 1;
    disp( [x feval(fh, x)] )
    myrel = abs((x-xold)/x);
end;
```

```
if myrel <= re
    conv = 1;
else
    conv = 0;
end;

f = feval(fh, x);
```

Try it out. Verify that you can call `newtfun` with less than three output variables. Also check `help newton`.

A function handle gives you more than just a reference to the function. See **Using MATLAB: Programming and Data Types: Function Handles**.

Functions, such as `feval`, `fplot`, `newtfun`, etc. which take function handles as arguments are referred to by MATLAB as *function functions*, as opposed to functions which take numeric arrays as arguments.

Use of a function handle to evaluate a function supersedes the earlier use of `feval` where a string containing the function's name was passed as an argument.

10.4 Command/function duality

In the earliest versions of MATLAB there was a clear distinction between *commands* like

```
clear
save junk x y z
whos
```

and *functions* like

```
sin(x)
plot(x, y)
```

If commands had any arguments they had to be separated by blanks with no brackets. Commands altered the environment, but didn't return results. New commands could not be created with M-files.

From Version 4 onwards commands and functions are 'duals', in that commands are considered to be functions taking string arguments. So

```
axis off
```

is the same as

```
axis( 'off' )
```

Other examples are

```
disp Error!
hold('on')
```

This duality makes it possible to generate command arguments with string manipulations, and also to create new commands with M-files.

10.5 Function name resolution

Remember that a variable in the workspace can 'hide' a built-in function of the same name, and a built-in function can hide an M-file.

Specifically, when MATLAB encounters a name it resolves it in the following steps:

1. Checks if the name is a variable.
2. Checks if the name is a *subfunction* of the calling function.
3. Checks if the name is a *private function*.
4. Checks if the name is in the directories specified by MATLAB's search path.

MATLAB therefore always tries to use a name as a variable first, before trying to use it as a script or function.

10.6 Debugging M-files

Run-time errors (as opposed to syntax errors) which occur inside function M-files are often hard to fix, because the function workspace is lost when the

error forces a return to the base workspace. The Editor/Debugger enables you to get inside a function while it is running, to see what's going wrong.

10.6.1 Debugging a script

To see how to debug interactively, let's first try the script `newtgen.m` in Section 10.3. Go through the following steps.

- Open `newtgen.m` with the MATLAB Editor/Debugger. (Incidentally, have you found out that you can run the Editor directly from the command line, e.g. with `edit newtgen`).

  You will have noticed that the lines in the Editor window are numbered. You can generate these line numbers from the command line for reference purposes with the command `dbtype`:

```
dbtype newtgen

1     % Newton's method in general
2     % exclude zero roots!
3
4     steps = 0;                          % iteration
                                            counter
5     x = input( 'Initial guess: ' );     % estimate
                                            of root
6     re = 1e-8;                          % required
                                            relative
                                            error
7     myrel = 1;
8
9     while myrel > re & (steps < 20)
      xold = x;
11      x = x - f(x)/df(x);
12      steps = steps + 1;
13      disp( [x f(x)] )
14      myrel = abs((x-xold)/x);
15    end;
16
17    if myrel <= re
18      disp( 'Zero found at' )
19      disp( x )
20    else
21      disp( 'Zero NOT found' )
22    end;
```

- To get into Debug mode you need to set a *breakpoint* just before where you think the problem might be. Alternatively, if you just want to 'step through'

a script line by line, set a breakpoint at the first executable statement. The column to the right of the line numbers is called the *breakpoint alley*. You can only set breakpoints at executable statements—these are indicated by dashes in the breakpoint alley.

Set a breakpoint at line 4 (`steps = 0;`) by clicking in the breakpoint alley. You can remove a breakpoint by clicking on the breakpoint icon, or using the Editor's **Breakpoints** menu (this menu also allows you to specify stopping conditions). You can also set/clear breakpoints on the current line with the set/clear breakpoint button on the toolbar.

- Having set your breakpoints, run the script in the Editor by clicking the run button in the toolbar, or with **Debug -> Run (F5)**.

  You can also run the script from the command line.

- When the script starts to run, there are two things in particular to notice.

  First, the symbol `K` appears to left of the command-line prompt to remind you that MATLAB is in debug mode.

  Second, a green arrow appears just to the right of the breakpoint in the Editor. The arrow indicates the next statement which is *about to be* executed.

- Now step through the script with **Debug -> Step (F10)**. Note when line 5 is executed you need to enter the value of `x` at the command line.

- When you get to line 11 (`x = x - f(x)/df(x);`) in this way, use **Debug -> Step In (F11)** and the Debugger will take you into the functions `f.m` and `df.m`.

- Continue with **F10**. Note that output appears in the Command Window as each `disp` statement is executed.

- There are a number of ways of examining variable values in debug mode:

  1. Position the cursor to the left of the variable in the Editor. Its current value appears in a box—this is called a datatip.
  2. Type the name of the variable in the Command Window.
  3. Use the Array Editor: open the Workspace browser and double-click a variable. If you arrange your windows carefully you can watch the value of a variable change in the Array Editor while you step through a program.

     Note that you can view variables only in the current workspace. The Editor has a **Stack** field to the right of the toolbar where you can select the workspace. For example, if you have stepped into `f.m` the current workspace is shown as `f`. At this point you can view variables in the `newtgen` workspace by selecting `newtgen` in the **Stack** field.

You can use the Array Editor or the command line to change the value of a variable. You can then continue to see how the script performs with the new value.

- Another useful debugging feature is **Debug -> Go Until Cursor**. This enables you to continue running the script to the line where you've positioned the cursor.
- To quit debugging click the exit debug mode button in the Editor/Debugger toolbar, or select **Debug -> Exit Debug Mode**.

  If you forget to quit debugging you won't be able to get rid of the `K` prompt on the command line!

10.6.2 Debugging a function

You can't run a function directly in the Editor/Debugger—you have to set a breakpoint in the function and run it from the command line. Let's use `newtfun.m` as an example.

- Open `newtfun.m` in the Editor/Debugger.
- Set a breakpoint at line 8 (`steps = 0;`).
- In the Command Window set up function handles for `f` and `df` and call `newtfun`:

```
fhand = @f;
dfhand = @df;
[x f conv] = newtfun(fhand, dfhand, 10)
```

- Note that MATLAB goes into debug mode and takes you to the breakpoint in `newtfun`. Now you can continue debugging as before.

Debugging may also be done from the command line with the debug functions. See `help debug`.

10.7 Recursion

Many (mathematical) functions are defined *recursively*, i.e. in terms of simpler cases of themselves, e.g. the factorial function may be defined recursively as

$$n! = n \times (n-1)!$$

as long as 1! is defined as 1. MATLAB allows functions to call themselves; this process is called *recursion*. The factorial function may be written recursively in an M-file `fact.m` like this:

```
function y = fact(n)
% FACT       Recursive definition of n!

if n > 1
  y = n * fact(n-1);
else
  y = 1;
end;
```

Recursive functions are usually written in this way: an `if` statement handles the general recursive definition; the `else` part handles the special case ($n = 1$).

Although recursion appears deceptively simple, it is an advanced topic, as the following experiment demonstrates. Insert the statement `disp(n)` into the definition of `fact` immediately *above* the `if` statement, and run `fact(5)` from the command line. The effect is what you might expect: the integers 5 to 1 in descending order. Now move `disp(n)` to *below* the `if` statement, and see what happens. The result is the integers 1 to 5 in *ascending* order now, which is rather surprising.

In the first case, the value of `n` is displayed each time `fact` is called, and the output is obvious enough. However, there is the world of difference between a recursive function being *called*, and *executed*. In the second case, the `disp` statement is only executed after the `if` has finished executing. And when is that exactly? Well, when the initial call to `fact` takes place, `n` has the value 5, so the first statement in the `if` is invoked. However, the value of `fact(4)` is not known at this stage, so a *copy* is made of all the statements in the function which will need to be executed once the value of `fact(4)` is known. The reference to `fact(4)` makes `fact` call itself, this time with a value of 4 for `n`. Again, the first statement in the `if` is invoked, and MATLAB discovers that it doesn't know the value of `fact(3)` this time. So another (different) copy is made of all the statements that will have to be executed once the value of `fact(3)` is known. And so each time `fact` is called, separate copies are made of all the statements *yet to be executed*. Finally, MATLAB joyfully finds a value of `n` (1) for which it actually knows the value of `fact`, so it can at last begin to execute (in reverse order) the pile of statements which have been damming up inside the memory.

This discussion illustrates the point that recursion should be treated with respect. While it is perfectly in order to use it in an example like this, it can chew up huge amounts of computer memory and time.

Summary

- Good structured programming requires real problem-solving programs to be broken down into function M-files.
- The name of a function in the function definition line should be the same as the name of the M-file under which it is saved. The M-file must have the extension `.m`.
- A function may have input and output arguments, which are usually its only way of communicating with the workspace. Input/output arguments are dummy variables (placeholders).
- Comment lines up to the first non-comment line in a function are displayed when `help` is requested for the function.
- Variables defined inside a function are local variables and are inaccessible outside the function.
- Variables in the workspace are inaccessible inside a function unless they have been declared `global`.
- A function does not have to have any output arguments.
- Input arguments have the appearance of being passed by value to a function. This means that changes made to an input argument inside a function are not reflected in the actual input argument when the function returns.
- A function may be called with fewer than its full number of input/output arguments.
- The functions `nargin` and `nargout` indicate how many input and output arguments are used on a particular function call.
- Variables declared `persistent` inside a function retain their values between calls to the function.
- Subfunctions in an M-file are accessible only to the primary function and to other subfunctions in the same M-file.
- Private functions are functions residing in a subdirectory named `private` and are accessible only to functions in the parent directory.
- Functions may be parsed (compiled) with the `pcode` function. The resulting code has the extension `.p` and is called a P-code file.
- The Profiler enables you to find out where your programs spend most of their time.

- A handle for a function is created with `@`.

  A function may be represented by its handle. In particular the handle may be passed as an argument to another function.
- `feval` evaluates a function whose handle is passed to it as an argument.
- MATLAB first tries to use a name as a variable, then as a built-in function, and finally as one of the various types of function.
- Command/function duality means that new commands can be created with function M-files, and that command arguments may be generated with string manipulations.
- The Editor/Debugger enables you to work through a script or function line-by-line in debug mode, examining and changing variables on the way.
- A function may call itself. This feature is called recursion.

EXERCISES

10.1 Change the function `stars` of Section 10.2 to a function `pretty` so that it will draw a line of any specified character. The character to be used must be passed as an additional input (string) argument, e.g. `pretty(6, '$')` should draw six dollar symbols.

10.2 Write a script `newquot.m` which uses the Newton quotient $[f(x+h)-f(x)]/h$ to estimate the first derivative of $f(x)=x^3$ at $x=1$, using successively smaller values of h: 1, 10^{-1}, 10^{-2}, etc. Use a function M-file for $f(x)$.

Rewrite `newquot` as a function M-file able to take a handle for $f(x)$ as an input argument.

10.3 Write and test a function `double(x)` which doubles its input argument, i.e. the statement `x = double(x)` should double the value in `x`.

10.4 Write and test a function `swop(x, y)` which will exchange the values of its two input arguments.

10.5 Write your own MATLAB function to compute the exponential function directly from the Taylor series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

The series should end when the last term is less than 10^{-6}. Test your function against the built-in function `exp`, but be careful not to make x too large—this could cause rounding error.

10.6 If a random variable X is distributed normally with zero mean and unit standard deviation, the probability that $0 \le X \le x$ is given by the standard normal function $\Phi(x)$. This is usually looked up in tables, but it may be approximated as follows:

$$\Phi(x) = 0.5 - r(at + bt^2 + ct^3),$$

where $a = 0.4361836$, $b = -0.1201676$, $c = 0.937298$,
$r = \exp(-0.5x^2)/\sqrt{2\pi}$, and $t = 1/(1 + 0.3326x)$.
Write a function to compute $\Phi(x)$, and use it in a program to write out its values for $0 \le x \le 4$ in steps of 0.1. Check: $\Phi(1) = 0.3413$.

10.7 Write a function

```
function [x1, x2, flag] = quad( a, b, c )
```

which computes the roots of the quadratic equation $ax^2 + bx + c = 0$. The input arguments `a`, `b` and `c` (which may take any values) are the coefficients of the quadratic, and `x1`, `x2` are the two roots (if they exist), which may be equal. See Figure 3.3 in Chapter 3 for the structure plan. The output argument `flag` must return the following values, according to the number and type of roots:

0: no solution ($a = b = 0$, $c \neq 0$);
1: one real root ($a = 0$, $b \neq 0$, so the root is $-c/b$);
2: two real or complex roots (which could be equal if they are real);
99: any x is a solution ($a = b = c = 0$).

Test your function on the data in Exercise 3.5.

10.8 The Fibonacci numbers are generated by the sequence

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

Can you work out what the next term is? Write a recursive function `f(n)` to compute the Fibonacci numbers F_0 to F_{20}, using the relationship

$$F_n = F_{n-1} + F_{n-2},$$

given that $F_0 = F_1 = 1$.

10.9 The first three Legendre polynomials are $P_0(x) = 1$, $P_1(x) = x$, and $P_2(x) = (3x^2 - 1)/2$. There is a general *recurrence* formula for Legendre polynomials, by which they are defined recursively:

$$(n + 1)P_{n+1}(x) - (2n + 1)xP_n(x) + nP_{n-1}(x) = 0.$$

Define a recursive function `p(n,x)` to generate Legendre polynomials, given the form of P_0 and P_1. Use your function to compute `p(2,x)` for a few values of `x`, and compare your results with those using the analytic form of $P_2(x)$ given above.

11

Vectors as arrays & *advanced data structures

The objectives of this chapter are:

- **A** To enable you to solve problems which involve working carefully with array subscripts.
- **B** To introduce you to:
  - Structures.
  - Cells and cell arrays.

In MATLAB an array is just another name for a vector. So why have a large part of a chapter on arrays, when we have already been using vectors for most of the book? It is helpful to talk about arrays (as opposed to vectors) when we are concerned with handling individual elements by means of their subscripts, rather than with the vector as a whole. Thus, in the first three sections of this chapter we look at a number of problems which are best solved by treating vectors as arrays, usually with the help of `for` loops.

In the last three sections of this chapter we deal with another topic. The topic is advanced data structures.

11.1 Update processes

In Chapter 8 (**Update processes**) we considered the problem of calculating the temperature of orange juice (OJ) in a can as it cools in a fridge. This is an example of an *update process*, where the main variable is repeatedly updated over a period of time. We now examine how to solve this problem more generally.

The OJ is initially at temperature 25°C when it is placed in the fridge, where the ambient temperature F is 10°. The standard way to solve such an update process is to break the time period up into a number of small steps, each of length dt. If T_i is the temperature at the beginning of step i, we can get T_{i+1} from T_i as follows:

$$T_{i+1} = T_i - K\,dt\,(T_i - F), \tag{11.1}$$

where K is a physical constant, and units are chosen so that time is in minutes.

11.1.1 Unit time steps

We first solve the problem using a unit time step, i.e. $dt = 1$. The simplest way is to use scalars for the time and the temperature, as we saw in Chapter 8 (although we didn't use a unit time step there):

```
K = 0.05;
F = 10;
T = 25;                      % initial temperature of OJ

for time = 1:100             % time in minutes
  T = T - K * (T - F);       % dt = 1
  if rem(time, 5) == 0
    disp( [time T] )
  end
end;
```

Note the use of `rem` to display the results every 5 minutes: when `time` is an integer multiple of 5 its remainder when divided by 5 will be zero.

While this is undoubtedly the easiest way of writing the script, we cannot easily plot the graph of temperature against time. In order to do that, `time` and `T` must be vectors. The index of the `for` loop must be used as the subscript of each element of `T`. Here's the script (`update1.m`):

```
K = 0.05;
F = 10;
time = 0:100;              % initialize vector time
T = zeros(1,101);          % pre-allocate vector T
T(1) = 25;                 % initial temperature of OJ

for i = 1:100                        % time in minutes
  T(i+1) = T(i) - K * (T(i) - F);    % construct T
end;
```

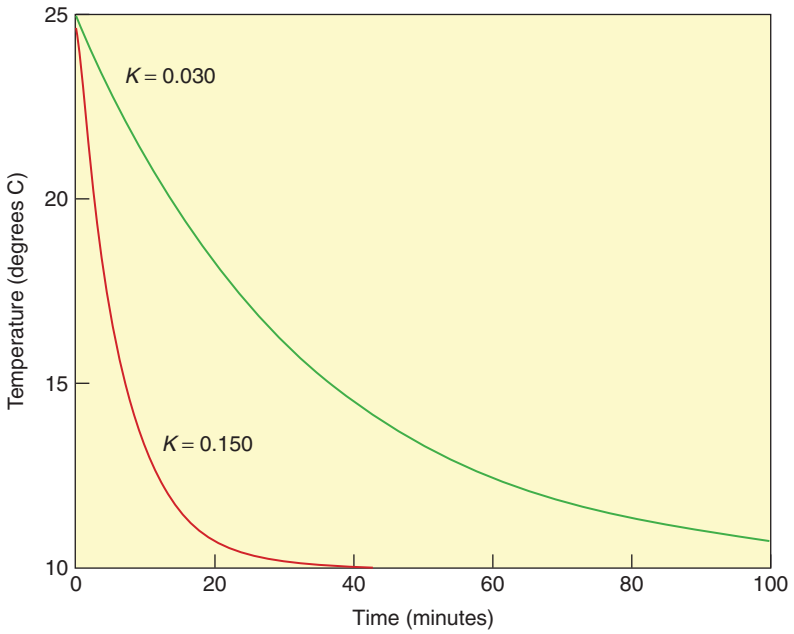


Figure 11.1 Cooling curves

```
disp([ time(1:5:101)' T(1:5:101)' ]);  % display results
plot(time, T), grid                    % every 5 mins
```

See Figure 11.1 for typical graphs.

Note:

- The statement `time = 0:100` sets up a (row) vector for time where `time(1)` has the value 0 minutes, and `time(101)` has the value 100 minutes. This is necessary because the first subscript of a MATLAB vector must be 1.
- The statement `T = zeros(1,101)` sets up a corresponding (row) vector for the temperature, with every element initialized to zero (again there must be 101 elements, because the first element is for the temperature at time zero). This process is called *pre-allocation*. It serves two important purposes.

  (a) Firstly, it clears a vector of the same name left over from a previous run. This could cause a conflict when attempting to display or plot `T` against `time`, if the vectors have different sizes. To see this, run `update1` as it stands. It should work perfectly. Now suppose

you decide to do the calculations over a shorter time period, say 50 minutes. Remove the `zeros` statement, and make the following additional two changes and re-run the script (but *don't* clear the workspace):

```
time = 0:50;     % initialize vector time
...

for i = 1:50     % time in minutes
```

This time you get an error message:

```
??? Error using ==> plot
Vectors must be the same lengths.
```

`whos` will reveal that `time` is correctly 51-by-1, but `T` is still 101-by-1. The `plot` command naturally requires these vectors to have the same lengths.

The problem is that, while the operation `0:50` redefines `time` correctly, the `for` loop does not have the same effect on `T`. Since `T` is being updated element by element, the unused elements from 51 to 101 from the previous run are left untouched in the workspace. Pre-allocation of the correct number of elements to `T` with `zeros` avoids this problem.

(b) Secondly, although the script will work without the `zeros` statement, as we have seen, it will be much *slower*, since then `T` has to be *redimensioned* during each repeat of the `for` loop, in order to make space for a new element each time.

It's extremely instructive to do an experiment using a vector size of 10000 elements, say, i.e.

```
time = 0:9999;            % initialize vector time
T = zeros(1,10000);       % pre-allocate vector T
...
for i = 1:10000
...
```

(and comment out the `disp` and `plot` statements, since these will obscure the issue). My Pentium II takes 0.99 seconds to run the script with pre-allocation of `T`, but 13.19 seconds to run without pre-allocation—more than 10 times longer. This could be a critical consideration in a script which does a lot of such element-by-element processing.

➤ The first element of `T` is set to the initial temperature of the OJ. This is the temperature at time zero.

- The `for` loop computes the values for `T(2),...,T(101)`.

  This arrangement ensures that temperature `T(i)` corresponds to `time(i)`.

- The colon operator is used to display the results at 5-minute intervals.

11.1.2 Non-unit time steps

It's not always appropriate and/or accurate enough to take $dt = 1$ in Equation (11.1). There is a standard way of generating the solution vector in MATLAB, given (almost) any value of dt. We introduce a more general notation to do this.

Call the initial time a, and the final time b. If we want time steps of length dt, the number m of such steps will be

$$m = (b - a)/dt.$$

The time at the *end* of step i will therefore be $a + i\,dt$.

The following script, `update2.m`, implements this scheme. It has some additional features. It prompts you for a value of dt, and checks that this value gives an integer number of steps m. It also asks you for the output interval `opint` (the intervals in minutes at which the results are displayed in table form) and checks that this interval is an integer multiple of dt. Try the script with same sample values, e.g. $dt = 0.4$ and `opint` $= 4$.

```
K = 0.05;
F = 10;
a = 0;                  % initial time
b = 100;                % final time
load train
dt = input( 'dt: ' );
opint = input( 'output interval (minutes): ' );
if opint/dt ~= fix(opint/dt)
  sound(y, Fs)
  disp( 'output interval is not a multiple of dt!' )
  break
end;

m = (b - a) / dt;       % m steps of length dt
if fix(m) ~= m          % make sure m is integer
  sound(y, Fs)
  disp( 'm is not an integer - try again!' );
```

```
  break
end;

T = zeros(1,m+1);       % pre-allocate (m+1) elements
time = a:dt:b;
T(1) = 25;              % initial temperature

for i = 1:m
  T(i+1) = T(i) - K * dt * (T(i) - F);
end;

disp( [time(1:opint/dt:m+1)' T(1:opint/dt:m+1)'] )
plot(time, T),grid
```

Note:

- The vectors `T` and `time` must each have $m+1$ elements, because there are m time steps, and we need an extra element for the initial value of each vector.
- The expression `opint/dt` gives the index increment for displaying the results, e.g. $dt = 0.1$ and `opint` $= 0.5$ displays every (0.5/0.1)th element, i.e. every 5th element.

11.1.3 Using a function

The really cool way of solving this problem is to write a function to do it. This makes it a lot easier to generate a table of results for different values of *dt*, say, using the function from the command line. Here is `update2.m` rewritten as a function `cooler.m`:

```
function [time, T, m] = cooler( a, b, K, F, dt, T0 )

m = (b - a) / dt;       % m steps of length dt
if fix(m) ~= m          % make sure m is integer
  disp( 'm is not an integer - try again!' );
  break
end;

T = zeros(1,m+1);       % pre-allocate
time = a:dt:b;
T(1) = T0;              % initial temperature
```

```
for i = 1:m
  T(i+1) = T(i) - K * dt * (T(i) - F);
end;
```

Suppose you want to display a table of temperatures against time at 5 minute intervals, using $dt = 1$ and $dt = 0.1$. Here is how to do it (in the Command Window):

```
dt = 1;
[t T m] = cooler(0, 100, 0.05, 10, dt, 25);
table(:,1) = t(1:5/dt:m+1)';
table(:,2) = T(1:5/dt:m+1)';
dt = 0.1;
[t T m] = cooler(0, 100, 0.05, 10, dt, 25);
table(:,3) = T(1:5/dt:m+1)';
format bank
disp(table)
```

Output:

```
     0       25.00       25.00
  5.00       21.61       21.67
 10.00       18.98       19.09
   ...
100.00       10.09       10.10
```

Note:

- The advantage of using a function which generates a vector output variable is that even if you forget to pre-allocate the vector inside the function (with `zeros`) MATLAB automatically clears any previous versions of the output vector before returning from the function.
- The variable `table` is a two-dimensional array (or matrix). Recall that the colon operator may be used to indicate all elements of a matrix in a row or column. So `table(:,1)` means the elements in every row and column 1, i.e. the entire first column. The vectors `t` and `T` are row vectors, so they must be transposed before being inserted into columns 1 and 2 of `table`.

  The third column of `table` is inserted in a similar way.
- The results in the third column (for $dt = 0.1$) will be more accurate.

11.1.4 Exact solution

This cooling problem has an exact mathematical solution. The temperature $T(t)$ at time t is given by the formula

$$T(t) = F + (T_0 - F)e^{-Kt}, \tag{11.2}$$

where T_0 is the initial temperature. You can insert values for this exact solution into a fourth column of `table`, by vectorizing the formula, as follows:

```
tab(:,4) = 10 + (T(1)-10)*exp(-0.05 * t(1:5/dt:m+1)');
```

The enlarged table should look something like this:

```
    0       25.00       25.00       25.00
 5.00       21.61       21.67       21.68
10.00       18.98       19.09       19.10
  ...
```

Note that the numerical solution generated by Equation (11.1) gets more accurate as *dt* gets smaller. That is because Equation (11.2) (the exact solution) is derived from Equation (11.1) *in the limit* as $dt \to 0$.

EXERCISE

To complete the analysis of the cooling problem, you may like to superimpose graphs for different values of *K* and draw them in different colors. It would also be useful to label each graph at some convenient place, e.g. with the label `'K = 0.08'`. The command

```
gtext( 'text' )
```

causes an arrow to appear on the graphics screen. You can position the arrow with a mouse or the arrow keys. When you click (or press **Enter**), the specified text appears at that point.

The function `gtext` takes a string argument. If you want to label with a variable numeric value, use `sprintf` to convert a number into a string. It works just like `fprintf`, except that the output goes into a string variable, e.g.

```
gtext( sprintf('K = %5.3f', K) )
```

- Draw some graphs, on the same set of axes, for different values of K, in different colors, and label them. Figure 11.1 shows what to aim for.

Also superimpose the exact solution, given by Equation (11.2).

Plots can also be labeled interactively in plot editing mode or with the Properties Editor.

11.2 Frequencies, bar charts and histograms

11.2.1 A random walk

Imagine an ant walking along a straight line, e.g. the x-axis. She starts at $x = 40$. She moves in steps of one unit along the line. Each step is to the left or the right with equal probability. We would like a visual representation of how much time she spends at each position.

Start by running the following script, `ant.m`:

```
f = zeros(1,100);
x = 40;

for i = 1:1000
  r = rand;
  if r >= 0.5
    x = x + 1;
  else
    x = x - 1;
  end
  f(x) = f(x) + 1;
end
```

Now enter the statement `bar(f)` in the Command Window. You should get a graph which is similar to the one in Figure 11.2.

Note:

- The function `rand` returns a random number in the range 0–1. If it's greater than 0.5, the ant moves right (`x = x + 1`), otherwise she moves left (`x = x - 1`).
- The vector `f` has 100 elements, initially all zero. We define `f(x)` as the number of times the ant lands at position `x`. Suppose her first step is to the right, so `x` has the value 41. The statement

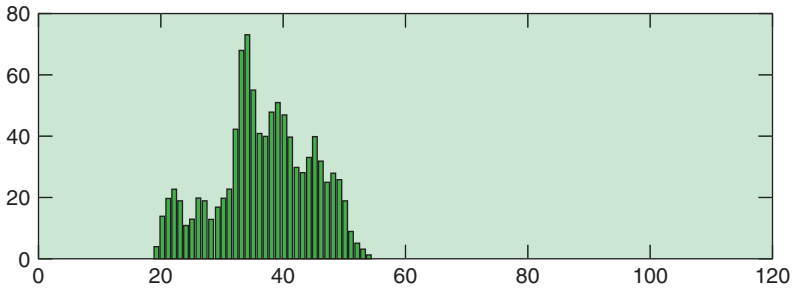


Figure 11.2 Random walk frequencies

```
f(x) = f(x) + 1
```

then increases the value of `f(41)` to 1, meaning that she has been there once. When she next wanders past this value of `x`, `f(41)` will be increased to 2, meaning she's been there twice.

When I ran this script, the final value of `f(41)` was 33—the number of times the ant was there.

- `f(x)` is called a *frequency distribution*, and the graph obtained from `bar(f)` is called a *bar graph*. Each element of `f` is represented by a vertical bar of proportional height. See also `help bar`.
- The script `ant.m` *simulates* the random movement of the ant. If you rerun it, you will get a different bar graph, because `rand` will generate a different sequence of random numbers. Simulation is discussed more fully in Chapter 15.

11.2.2 Histograms

Another helpful way of representing data is with a *histogram*.

As an example, suppose 12 students write a test and obtain the following marks (expressed as percentages), which are assigned to the vector `m`:

```
0   25   29   35   50   55   55   59   72   75   95   100
```

The statement `hist(m)` will draw a histogram. Try it. The histogram shows the distribution of marks in 10 'bins' (categories) which are equally spaced between the minimum (0) and maximum (100) marks in the set of data. The number of bins (10 by default) can be specified by a second argument, e.g. `hist(m, 25)`.

To generate the frequences plotted by `hist` use the following form (which does not actually draw the histogram):

```
[n x] = hist(m)
```

`n` is a vector containing the frequences, viz.,

```
1    0    2    1    0    4    0    2    0    1    1
```

For example, there is one mark in the first bin (0–9), none in the second (10–19), two in the third, etc.

The second output vector `x` contains the mid-points of the bins, such that `bar(x, n)` plots the histogram. See Help for more details.

Note the subtle difference between a histogram and a bar graph. The values plotted by `hist` are computed from the distribution of values in a vector, whereas `bar` generates a bar graph directly from the values themselves.

11.3 *Sorting

One of the standard applications of arrays is sorting a list of numbers into, say, ascending order. Although MATLAB has its own sorting function (`sort`), you may be interested in how sorting algorithms actually work.

11.3.1 Bubble Sort

The basic idea is that the unsorted list is assigned to a vector. The numbers are then ordered by a process which essentially passes through the vector many times, swopping consecutive elements that are in the wrong order, until all the elements are in the right order. Such a process is called a Bubble Sort, because the smaller numbers rise to the top of the list, like bubbles of air in water. (In fact, in the version shown below, the largest number will 'sink' to the bottom of the list after the first pass, which really makes it a 'Lead Ball' sort.) There are many other methods of sorting, such as the Quick Sort, which may be found in most textbooks on computer science. These are generally more efficient than the Bubble Sort, but its advantage is that it is by far the easiest method to program. A structure plan for the Bubble Sort is as follows:

1. Input the list *X*
2. Set *N* to the length of *X*

Table 11.1 Memory during a Bubble Sort

	1st pass	2nd pass	3rd pass	4th pass
X_1:	27/13	13/9	9/5	5/3
X_2:	13/27/9	9/13/5	5/9/3	3/5
X_3:	9/27/5	5/13/3	3/9	9
X_4:	5/27/3	3/13	13	13
X_5:	3/27 4 tests	27 3 tests	27 2 tests	27 1 test

3. Repeat $N-1$ times with counter K:
   Repeat $N-K$ times with counter J:
   If $X_j > X_{j+1}$ then
   Swop the contents of X_j and X_{j+1}
4. Stop since the list X is now sorted.

As an example, consider a list of five numbers: 27, 13, 9, 5 and 3. They are initially input into the vector X. Part of MATLAB's memory for this problem is sketched in Table 11.1. Each column shows the list during each *pass*. A stroke in a row indicates a change in that variable during the pass as the script works down the list. The number of tests ($X_j > X_{j+1}$?) made on each pass is also shown in the table. Work through the table by hand with the structure plan until you understand how the algorithm works.

Sorting algorithms are compared by calculating the number of tests (comparisons) they carry out, since this takes up most of the execution time during the sort. On the Kth pass of the Bubble Sort there are exactly $N-K$ tests, so the total number of tests is

$$1+2+3+\cdots+(N-1)=N(N-1)/2$$

(approximately $N^2/2$ for large N). For a list of five numbers there are therefore 10 tests, but for 10 numbers there are 45 tests. The computer time needed goes up as the square of the length of the list.

The function M-file `bubble.m` below departs slightly from the structure plan above, which will make $N-1$ passes, *even if the list is sorted before the last pass*. Since most real lists are partially sorted, it makes sense to check after

each pass if any swops were made. If none were, the list must be sorted, so unnecessary (and therefore time-wasting) tests can be eliminated. In the function, the variable `sorted` is used to detect when the list is sorted, and the outer loop is coded instead as a non-deterministic `while` loop. Here it is:

```
function y = bubble( x )
n = length(x);
sorted = 0;                % flag to detect when sorted
k = 0;                     % count the passes

while ~sorted
  sorted = 1;              % they could be sorted
  k = k + 1;               % another pass
  for j = 1:n-k            % fewer tests on each pass
    if x(j) > x(j+1)       % are they in order?
      temp = x(j);         % no ...
      x(j) = x(j+1);
      x(j+1) = temp;
      sorted = 0;          % a swop was made
    end
  end
end;

y = x;
```

You can test it on the command line to sort say 20 random numbers as follows:

```
r = rand(1,20);
r = bubble( r );
```

Note how `bubble` changes its input vector.

On my PC `bubble` takes 1.81 seconds to sort 200 random numbers, but 7.31 seconds to sort 400 numbers. This is consistent with the theoretical result obtained above.

11.3.2 MATLAB's `sort`

The built-in MATLAB function `sort` returns two output arguments: the sorted list (in ascending order) and a vector containing the *indexes* used in the sort, i.e. the positions of the sorted numbers in the original list. If the random numbers

```
r =   0.4175    0.6868    0.5890    0.9304    0.8462
```

are sorted with the command

```
[y,i] = sort(r)
```

the output variables are

```
y =  0.4175    0.5890    0.6868    0.8462    0.9304
i =  1     3     2     5     4
```

For example, the index of the second largest number (0.5890) is 3, which is its subscript in the original unsorted list `r`.

As a matter of fact, the built-in functions `max` and `min` also return second output variables giving indexes.

MATLAB's `sort` is *very* fast. My PC takes 2.36 seconds to sort a list of one million random numbers! This is because (a) a Quick Sort is used, and (b) the script has been compiled as a built-in function; this makes for faster code.

11.4 *Structures

Up to now we have seen arrays with only one type of element—all numeric, or all character. A MATLAB *structure* allows you to put different kinds of data in its various *fields*. For example, we can create a structure called `student` with one field for a student's name,

```
student.name = 'Thandi Mangwane';
```

a second field for her student ID number,

```
student.id = 'MNGTHA003';
```

and a third field for all her marks to date,

```
student.marks = [36 49 74];
```

To see the whole structure, enter its name:

```
student

student =
     name: 'Thandi Mangwane'
```

```
        id: 'MNGTHA003'
    marks: [36 49 74]
```

Here's how to access her second mark:

```
student.marks(2)
ans =
    49
```

Note the use of the dot to separate the structure name from its fields when creating it and when accessing its fields.

To add further elements to the structure, use subscripts after the structure name:

```
student(2).name = 'Charles Wilson'
student(2).id = 'WLSCHA007'
student(2).marks = [49 98]
```

(the original student, Thandi Mangwane, is now accessed as `student(1)`). Note that field sizes do not have to conform across elements of a structure array: `student(1).marks` has three elements, while `student(2).marks` has only two elements.

The `student` structure now has size 1-by-2: it has two elements, each element being a student with three fields. Once a structure has more than one element MATLAB does not display the contents of each field when you type the structure name at the command line. Instead, it gives the following summary:

```
student

student =
1x2 struct array with fields:
    name
    id
    marks
```

You can also use `fieldnames(student)` to get this information.

A structure array can be pre-allocated with the `struct` function. See Help.

A structure field can contain any kind of data, even another structure—why ever not? Thus we can create a structure `course` of courses taken by students,

where one field is the name of the course, and another field is a `student` structure with information on all the students taking that particular course:

```
course.name = 'MTH101';
course.class = student;
course

course =
     name: 'MTH101'
    class: [1x2 struct]
```

We can set up a second element of `course` for a different class of students:

```
course(2).name = 'PHY102';
course(2).class = ...
```

To see all the courses:

```
course(1:2).name
ans =
MTH101
ans =
PHY102
```

To see all the students in a particular course:

```
course(1).class(1:2).name
ans =
Thandi Mangwane
ans =
Charles Wilson
```

There is a curious function called `deal` which 'deals inputs to outputs'. You can use it to generate 'comma-separated variable lists' from structure fields:

```
[name1, name2] = deal(course(1).class(1:2).name);
```

(but you don't actually need the commas here ...).

You can use the `rmfield` function to remove fields from a structure.

11.5 *Cell arrays

A *cell* is the most general data object in MATLAB. A cell may be thought of as a 'data container', which can contain any type of data: numeric arrays, strings, structures, or cells. An array of cells (and they almost always occur in arrays) is called a *cell array*. While you might think a cell sounds the same as a structure, a cell is more general, and there are also notational differences (which are confusing!).

11.5.1 Assigning data to cell arrays

There are a number of ways of assigning data to cell arrays.

- Cell indexing:

```
c(1,1) = {rand(3)};
c(1,2) = {char('Bongani', 'Thandeka')};
c(2,1) = {13};
c(2,2) = {student};

c =
    [3x3 double]    [2x8 char  ]
    [        13]    [1x2 struct]
```

(assuming that the structure `student` created above still exists). Here the round brackets on the left-hand side of the assignments refer in the normal way to the elements of the cell array. What is different are the curly braces on the right. Curly braces indicate the *contents* of a cell; on the right-hand side of an assignment they are technically cell array *constructors* (remember that each element of this array is a cell).

So read the first statement as

'Construct a cell containing `rand(3)` and assign the cell to element `1,1` of the cell array `c`.'

- Content indexing:

```
c{1,1} = rand(3);
c{1,2} = char('Bongani', 'Thandeka');
c{2,1} = 13;
c{2,2} = student;
```

Here the curly braces on the left indicate the *contents* of the cell element at that particular location. So read the first statement as

'The contents of the cell at location `1,1` becomes `rand(3)`.'

- You can use the curly braces to construct an entire cell array in one statement:

```
b = {[1:5], rand(2); student, char('Jason', 'Amy')}
b =
    [1x5 double]    [2x2 double]
    [1x2 struct]    [2x5 char  ]
```

  A cell may contain another cell array; nested curly braces may be used to create nested cell arrays.

- The `cell` function enables you to pre-allocate empty cell arrays, e.g.

```
a = cell(3,2)      % empty 3-by-2 cell array
a =
     []      []
     []      []
     []      []
```

  You can then use assignment statements to fill the cells, e.g.

```
a(2,2) = {magic(3)}
```

Note: If you already have a numeric array with a certain name, don't try to create a cell array of the same name by assignment without first clearing the numeric array. If you don't clear the numeric array, MATLAB will generate an error (it will think you are trying to mix cell and numeric syntaxes).

11.5.2 Accessing data in cell arrays

You can access cell contents using content indexing (curly braces):

```
r = c{1,1}
r =
    0.4447    0.9218    0.4057
    0.6154    0.7382    0.9355
    0.7919    0.1763    0.9169
```

To access a subset of a cell's contents you can concatenate curly braces and round brackets if necessary:

```
rnum = c{1,1}(2,3)
rnum =
    0.9355
```

Here the curly braces (content indexing) indicate the contents of the cell array element `c(1,1)`, which is a 3-by-3 numeric matrix. The subscripts `(2,3)` then indicate the appropriate element within the matrix.

Curly braces may be concatenated to access nested cell arrays.

11.5.3 Using cell arrays

Cell arrays come into their own when you need to access (different types of) data as 'comma-separated variable lists', as the next example demonstrates.

The functions `varargin` and `varargout`, which allow a function to have any number of input or output arguments, are none other than cell arrays. The function `testvar` has a variable number of input arguments, which are doubled into the variable number of output arguments (assuming that the number of output arguments does not exceed the number of input arguments):

```
function [varargout] = testvar(varargin)

for i = 1:length(varargin)
    x(i) = varargin{i};        % unpack the input args
end

for i = 1:nargout             % how many output arguments?
    varargout{i} = 2*x(i);     % pack up the output args
end
```

Command line:

```
[a b c] = testvar(1, 2, 3, 4)
a =
     2
b =
     4
c =
     6
```

When a function is called with the input argument `varargin` MATLAB automatically packs the list of corresponding input arguments into a cell array. You then simply unpack the cell array inside the function using the curly brace content indexing. You similarly pack up the output arguments into the cell array `varargout`.

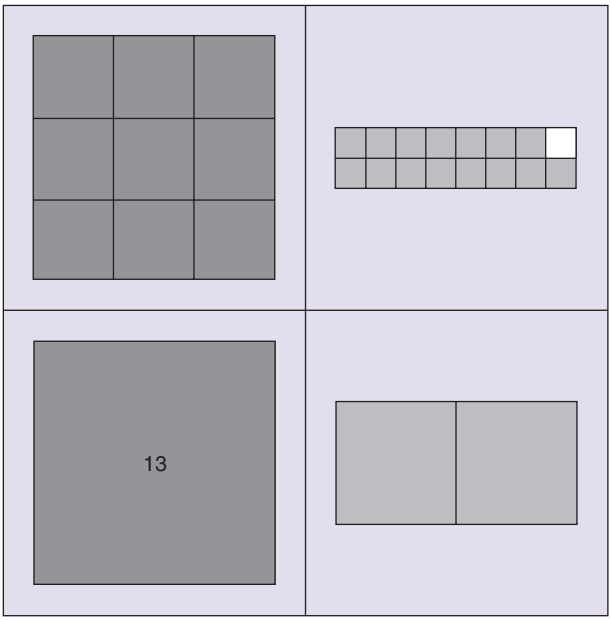


Figure 11.3 Result of `cellplot(c)`

Note that, should your function have some compulsory input and output arguments, `varargin` and `varargout` must appear at the end of their respective argument lists.

MATLAB has a discussion on when to use cell arrays in the section **Organizing Data in Cell Arrays**, to be found in **Using MATLAB: Programming and Data Types: Structures and Cell Arrays**.

11.5.4 Displaying and visualizing cell arrays

The function `celldisp` recursively displays the contents of a cell array.

The function `cellplot` draws a visualization of a cell array. Figure 11.3 depicts the contents of the cell array `c`, created above. Non-empty array elements are shaded.

11.6 *Classes and objects

You'd have to be living face down in a moon crater not to have heard about object-oriented programming. *Tom Swan*

> Object-oriented programming is an exceptionally bad idea which could only have originated in California. *Edsger Dijkstra*

MATLAB, along with most other modern programming languages, espouses the cause of object-oriented programming, and has all the usual paraphernalia associated with this paradigm: classes, objects, encapsulation, inheritance, operator overloading, etc., etc.

Object-oriented programming is a subject which requires an entire book to do justice to it. If you want to learn about the concepts, there are many excellent books on the subject. If you want to see how MATLAB implements object-oriented programming, consult the online Help: **Using MATLAB: Programming and Data Types: MATLAB Classes and Objects**.

Summary

- A MATLAB structure allows you to store different types of data in its various fields.
- Arrays of structures may be created.
- A cell is the most general data object in MATLAB, and can store any type of data.
- An array of cells is called a cell array—each cell in the array can store different types of data, including other cell arrays.
- A cell array is constructed with curly braces `{}`.
- The contents of cell in a cell array is accessed with content indexing (curly braces).
- Cell elements in a cell array are accessed in the usual way with round brackets (cell indexing).
- Arguments of functions using a variable number of arguments are packed into cell arrays.
- The function `cellplot` gives a visualization of a cell array.
- MATLAB implements object-oriented programming.

12

*More graphics

The objectives of this chapter are to introduce you to:

- **Handle Graphics;**
- **editing plots;**
- **animation;**
- **saving and exporting graphs;**
- **color, lighting and the camera.**

12.1 Handle Graphics

The richness and power of MATLAB graphics is made possible by its *Handle Graphics objects*. There is a very helpful section devoted to Handle Graphics in the online documentation **Using MATLAB: Graphics**. What follows is of necessity a brief summary of its main features.

Handle Graphics objects are the basic elements used in MATLAB graphics. The objects are arranged in a parent–child inheritance structure as shown in Figure 12.1. For example, Line and Text objects are children of Axes objects. This is probably the most common parent–child relationship. Axes objects define a region in a figure window and orient their children within this region. The actual plots in an Axes object are Line objects. The axis labels and any text annotations are Text objects. It is important to be aware of this parent–child hierarchy when you want to manipulate graphics objects using their handles.

Well, what exactly is the handle of a graphics object? Whenever MATLAB creates a graphics object it automatically creates a handle to that object. You can get the handle of the object using a function that explicitly returns the handle of a graphics object, or you can create the handle explicitly when you draw the

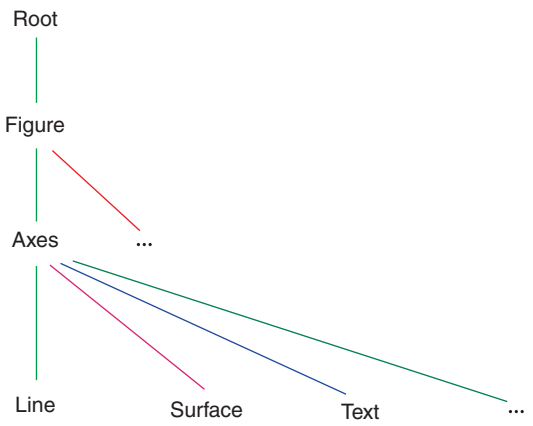


Figure 12.1 Parent–child relationships of Handle Graphics objects (from top to bottom)

graphics object. The handle itself has a floating point representation, but its actual value need not concern you. What is more important is to save its name and then to use the handle to change or manipulate your graphics object.

The root object is the only object whose handle is 0. There is only one root object, created by MATLAB at startup. All other objects are its descendants, as you can see in Figure 12.1.

12.1.1 Getting handles

Here's how to get handles:

- The functions that draw graphics objects can also be used to return the handle of the object drawn, e.g.

```
x = 0:pi/20:2*pi;
hsin = plot(x, sin(x))
hold on
hx = xlabel('x')
```

`hsin` is the handle of the Line object (the sine graph), and `hx` is the handle of the Text object (the *x* axis label) in the current Axes object.

The command

```
figure(h)
```

where `h` is an integer, creates a new figure, or makes figure `h` the current figure. `h` is the handle of the figure object.

- There are three functions that return the handle of particular graphics objects:

  `gcf` gets the handle of the current figure, e.g.

  ```
  hf = gcf;
  ```

  `gca` gets the handle of the current axes.

  `gco` gets the handle of the current graphics object, which is the last graphics object created or clicked on. For example, draw the sine graph above and get its handle `hsin`. Click on the graph in the figure window. Then enter the command

  ```
  ho = gco
  ```

  in the Command Window. `ho` should be set to the handle of the sine graph (it should have the same numeric value as `hsin`).

12.1.2 Graphics object properties and how to change them

Once you have the handle of a graphics object you can use it to change the properties of that object. As an example, draw a sine graph and get its handle, as demonstrated above:

```
x = 0:pi/20:2*pi;
hsin = plot(x, sin(x))
```

Now suppose you want to make the graph much thicker. Enter the following command:

```
set(hsin, 'linewidth', 4);
```

You should get a nice fat sine curve!

`linewidth` is just one of the many properties of our graphics object. To see all the property names of an object and their current values use `get(h)` where `h` is the object's handle. In the case of our sine graph:

```
get(hsin)
  Color = [0 0 1]
  EraseMode = normal
  LineStyle = -
  LineWidth = [4]
  Marker = none
  MarkerSize = [6]
  MarkerEdgeColor = auto
  MarkerFaceColor = none
```

```
XData = [ (1 by 41) double array]
YData = [ (1 by 41) double array]
ZData = []

BeingDeleted = off
ButtonDownFcn =
Children = []
Clipping = on
CreateFcn =
DeleteFcn =
BusyAction = queue
HandleVisibility = on
HitTest = on
Interruptible = on
Parent = [100.001]
Selected = off
SelectionHighlight = on
Tag =
Type = line
UIContextMenu = []
UserData = []
Visible = on
```

You can change any property value with the `set` function:

`set(`*handle, 'PropertyName', PropertyValue*`)`

The command `set(`*handle*`)` lists all the possible property values (where appropriate).

You can get an object's handle and change its properties all in the same breath. For example,

```
set(gcf, 'visible', 'off')
```

makes the current figure invisible (without closing it—i.e. it's still 'there'). No prizes for guessing how to make it visible again …

Property names are not case-sensitive, and you can abbreviate them to as few letters as make them unique. For example, you can abbreviate the `type` property to `ty`:

```
get(hsin,'ty')
ans =
line
```

(helpful if you don't know what type of objects you are dealing with!).

The different types of graphics objects don't all have the same set of properties, although there are some properties which are common to all graphics objects, such as `children`, `parent`, `type`, etc.

12.1.3 A vector of handles

If a graphics object has a number of children the `get` command used with the `children` property returns a vector of the children's handles. Sorting out the handles is then quite fun, and demonstrates why you need to be aware of the parent–child relationships!

As an example, plot a continuous sine graph and an exponentially decaying sine graph marked with `o`'s in the same figure:

```
x = 0:pi/20:4*pi;
plot(x, sin(x))
hold on
plot(x, exp(-0.1*x).*sin(x), 'o')
hold off
```

Now enter the command

```
hkids = get(gca,'child')
```

You will see that a vector of handles with two elements is returned. The question is, which handle belongs to which plot? The answer is that the handles of children of the axes are returned in the *reverse order in which they are created*, i.e. `hkids(1)` is the handle of the exponentially decaying graph, while `hkids(2)` is the handle of the sine graph. So now let's change the markers on the decaying graph, and make the sine graph much bolder:

```
set(hkids(1), 'marker', '*')
set(hkids(2), 'linew', 4)
```

You should get the plots shown in Figure 12.2.

If you are desperate and don't know the handles of any of your graphics objects you can use the `findobj` function to get the handle of an object with a property value that uniquely identifies it. In the original version of the plots in Figure 12.2 the decaying plot can be identified by its `marker` property:

```
hdecay = findobj('marker', 'o' )
```

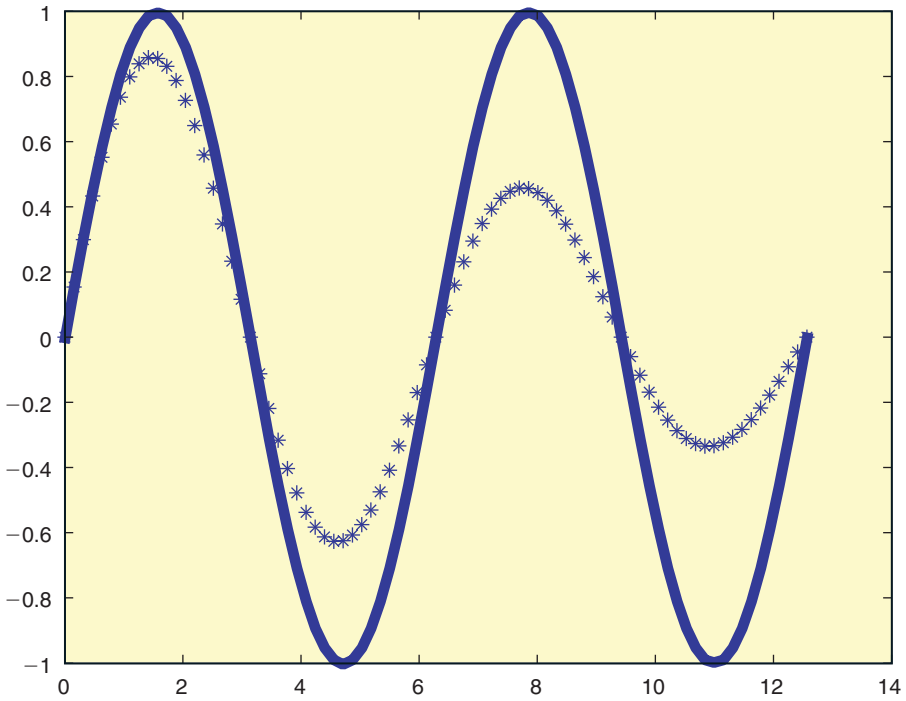


Figure 12.2 Result of manipulating a figure using the handles of axes children

12.1.4 Graphics object creation functions

Each of the graphics objects shown in Figure 12.1 (except the root object) has a corresponding creation function, named after the object it creates. See `help` for details.

12.1.5 Parenting

By default, all graphics objects are created in the current figure. However, you can specify the parent of an object when you create it. For example,

```
axes('Parent', figure_handle, ... )
```

creates an axes in the figure with handle *figure_handle*. You can also move an object from one parent to another by redefining its `parent` property:

```
set(gca, 'Parent', figure_handle)
```

12.1.6 Positioning figures

At startup MATLAB determines the default position and size of the figure window, according to the size of your computer screen. You can change this by setting the `Position` property of the figure object.

Before you tamper with the figure's `Position` property you need to know the dimensions of your screen—this is one of the root object's properties. For example,

```
get(0, 'screensize')
ans =
     1     1   800   600
```

i.e. my screen is 800 by 600 pixels. The units of `Screensize` are pixels by defaults. You can (naturally) change the units by setting the root's `Units` property. For example,

```
set(0, 'units', 'normalized')
```

normalizes the width and height of the screen to 1. This is useful when you are writing an M-file which is to be run on different computer systems.

Having sorted out the size of your screen, you can then fiddle with the figure `Position` property, which is defined as a vector

```
[left bottom width height]
```

`left` and `bottom` define the position of the first addressable pixel in the lower left corner of the window, specified with respect to the lower left corner of the screen. `width` and `height` define the size of the interior of the window (excluding the window border).

You can normalize the figure's `Unit` property as well as the root's `Unit` property. You can then position figures absolutely, without reference to variations in screen size. For example, the following code normalizes units and creates two figures in the upper half of the screen:

```
set(0, 'units', 'normalized')
h1 = figure('units', 'normalized', 'visible', 'off')
h2 = figure('units', 'normalized', 'visible', 'off')
```

```
set(h1, 'position', [0.05 0.5 0.45 0.35], 'visible', 'on')
set(h2, 'position', [0.53 0.5 0.45 0.35], 'visible', 'on')
```

Note that the `Visible` property is first set to `off` to avoid the figures being drawn in the default position when they are created. They are only drawn when their positions have been redefined. Alternatively you can do everything when the figures are created:

```
h1 = figure('un', 'normalized', 'pos',
[0.05 0.5 0.45 0.35])
```

12.2 Editing plots

There are a number of ways of editing plots.

12.2.1 Plot edit mode

To see how this works draw a graph, e.g. the friendly old sine. There are several ways to activate plot edit mode:

- Select **Tools -> Edit Plot** in the figure window.
- Click on the Edit Plot selection button in the figure window toolbar (arrow pointing roughly northwest).
- Run the `plotedit` command in the Command Window.

When a figure is in plot edit mode the toolbar selection button is highlighted. Once you are in plot edit mode select an object by clicking on it. Selection handles will appear on the selected object.

As an exercise, get the sine graph into plot edit mode and try the following:

- Select the graph (click on it). Selection handles should appear.
- Right-click on the selected object (the graph). A context menu appears.
- Use the context menu to change the graph's line style and color.
- Use the **Insert** menu to insert a legend (although this makes more sense where you have multiple plots in the figure).

- Insert a text box inside the figure close to the graph as follows. Click on the Insert Text selection button in the toolbar (the capital A). The cursor changes shape to indicate that it is in text insertion mode. Move the insertion point until it touches the graph somewhere and click. A text box appears. Enter some text in it.

  You can use a subset of TEX characters and commands in text objects. For example, the text x_k in Figure 17.1 was produced with the string `{\itx_k}`. See the entry `Text property` under `String[1]` in the online Help for a list of available TEX characters and commands.

- Having labeled the graph you can change the format of the labels. Select the label and right-click. Change the font size and font style.

- Play around with the Insert Arrow and Insert Line selection buttons on the toolbar to see if you can insert lines and arrows on the graph.

To exit plot edit mode, click the selection button or uncheck the **Edit Plot** option on the **Tools** menu.

12.2.2 Property Editor

The Property Editor is more general than plot edit mode. It enables you to change object properties interactively, rather than with the `set` function. It is ideal for preparing presentation graphics.

There are numerous ways of starting the Property Editor (you may already have stumbled onto some):

- If plot edit mode is enabled you can:
  - Double-click on an object.
  - Right-click on an object and select **Properties** from the context menu.
- Select **Figure Properties**, **Axes Properties** or **Current Object Properties** from the figure **Edit** menu.
- Run the `propedit` command on the command line.

To experiment with the Property Editor it will be useful to have multiple plots in a figure:

```
x = 0:pi/20:2*pi;
hsin = plot(x,sin(x))
hold on
hcos = plot(x,cos(x))
hold off
```

Start the Property Editor and work through the following exercises:

- The navigation bar at the top of the Property Editor (labeled **Edit Properties for:**) identifies the object being edited. Click on the down arrow at the right of the navigation bar to see all the objects in the figure. You will notice that there are two line objects. Immediately we are faced with the problem of identifying the two line objects in our figure. The answer is to give each of them tags by setting their `Tag` properties.

  Go back to the figure and select the sine graph. Back in the Property Editor the navigation bar indicates you are editing one of the line objects. You will see three tabs below the navigation bar: **Data**, **Style** and **Info**. Click the **Info** tab, and enter a label in the **Tag** box, e.g. `sine`. Press **Enter**. The tag `sine` immediately appears next to the selected line object in the navigation bar.

  Give the cosine graph a tag as well (start by selecting the other line object).

- Select the sine graph. This time select the **Style** tab and go berserk changing its color, line style, line width and markers.

- Now select the axes object. Use the **Labels** tab to insert some axis labels. Use the **Scale** tab to change the *y* axis limits, for example.

  Note that if you were editing a 3-D plot you would be able to use the **Viewpoint** tab to change the viewing angle and to set various camera properties.

Have fun!

12.3 Animation

There are three facilities for animation in MATLAB:

- The `comet` and `comet3` functions can be used to draw comet plots, as we mentioned briefly in Chapter 7.

- The `getframe` function may be used to generate 'movie frames' from a sequence of graphs. The `movie` function can then be used to play back the movie a specified number of times.

  The MATLAB online documention has the following script in **Using MATLAB: Graphics: Creating Specialized Plots: Animation**. It generates 16 frames from the Fast Fourier Transforms of complex matrices:

```
for k = 1:16
    plot(fft(eye(k+16)))
```

```
    axis equal
    M(k) = getframe;
end
```

Now play it back, say five times:

```
movie(M, 5)
```

You can specify the speed of the playback, among other things. See `help`.

- The most versatile (and satisfying) way of creating animations is by using the Handle Graphics facilities. Two examples follow.

12.3.1 Animation with Handle Graphics

For starters, run the following script, which should show the marker `o` tracing out a sine curve, leaving a trail behind it:

```
% animated sine graph
x = 0;
y = 0;
dx = pi/40;
p = plot(x, y, 'o', 'EraseMode', 'none');
   % 'xor' shows only current point
   % ' none' shows all points
axis([0 20*pi -2 2])

for x = dx:dx:20*pi;
    x = x + dx;
    y = sin(x);
    set(p, 'XData', x, 'YData', y)
    drawnow
end
```

Note:

- The statement

  ```
  p = plot(x, y, 'o', 'EraseMode', 'none');
  ```

  achieves a number of things. It plots the first point of the graph. It saves the handle `p` of the plot for further reference. And it specifies that the `Erasemode` property is `none`, i.e. the object must not be erased when it is drawn again. To achieve complete animation, set this property to `xor`—try it now. Then the object is erased each time it is redrawn (in a slightly different position), creating the classic animation effect.

- The statement

  ```
  set(p, 'XData', x, 'YData', y)
  ```

  sets the `x` and `y` data values of the object `p` to the new values generated in the `for` loop, and 'redraws' the object. However, it is not drawn on the screen immediately—it joins the 'event queue', where it waits until it is flushed out.

- Finally, the `drawnow` function flushes the event queue and draws the object on the screen so that we can see the fruit of our labors.

  As `help drawnow` will reveal, there are four events that flush the event queue:

  – a return to the MATLAB prompt—this is how you have seen all the graphs you've drawn up to now;

  – hitting a `pause` statement;

  – executing a `getframe` command;

  – executing a `drawnow` command.

  For example, you could make the marker move in a more stately fashion by replacing `drawnow` with `pause(0.05)`—that's 0.05 seconds.

The next example is based on one in the **Animation** section of the MATLAB documentation. It involves *chaotic* motion described by a system of three non-linear differential equations having a 'strange attractor' (known as the Lorenz strange attractor). The system can be written as

$$\frac{dy}{dt} = \mathbf{A}y,$$

where $y(t)$ is a vector with three components, and $\mathbf{A}$ is a matrix depending on y:

$$\mathbf{A}(y) = \begin{bmatrix} -8/3 & 0 & y(2) \\ 0 & -10 & 10 \\ -y(2) & 28 & -1 \end{bmatrix}.$$

The following script solves the system approximately using Euler's method (a more accurate numerical solution is discussed in Chapter 17), and shows the solution orbiting about two different attractors without settling into a steady orbit about either. Figure 12.3 shows the situation after a few thousand points have been plotted.

```
A = [ -8/3 0 0; 0 -10 10; 0 28 -1 ];
y = [35 -10 -7]';
h = 0.01;
```

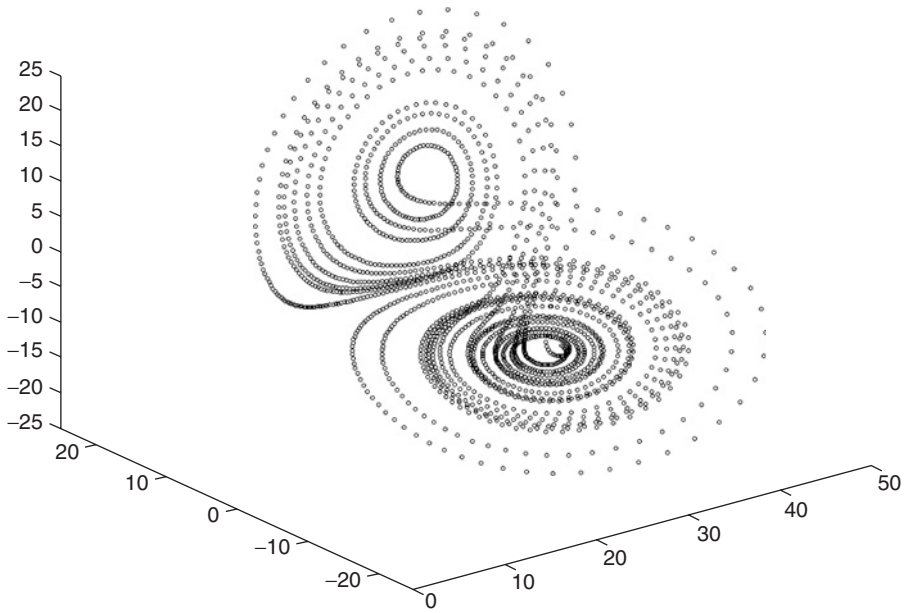


Figure 12.3 The Lorenz strange attractor

```
p = plot3(y(1), y(2), y(3), 'o', ...
    'erasemode', 'none', 'markersize', 2);
axis([0 50 -25 25 -25 25])
hold on
i = 1;

while 1
   A(1,3) = y(2);
   A(3,1) = -y(2);
   ydot = A*y;
   y = y + h*ydot;
   % Change color occasionally
   if rem(i,500) == 0
      set(p, 'color', [rand, rand, rand])
   end
   % Change coordinates
   set(p, 'XData', y(1), 'YData', y(2), 'ZData', y(3))
   drawnow
   i=i+1;
end
```

If all the points are all plotted in the same color you eventually wouldn't be able to see the new points generated: a large area of the screen would be filled with

the drawing color. The color is therefore set randomly after every 500 points are plotted.

12.4 Color etc.

12.4.1 Colormaps

MATLAB graphics can generate a rich variety of colors. You may be wondering how.

The following script shows a view of the Earth from space:

```
load earth
image(X); colormap(map)
axis image
```

(`axis image` is the same as `axis equal` except that the plot box fits tightly around the data). For a variation use `hot` as the argument of `colormap` instead of `map`.

The matrix `X` loaded from `earth` is 257-by-250. Each of the elements of `X` is an integer in the range 1–64. For example, here is a 3-by-3 submatrix of `X` (somewhere in northeast Africa):

```
X(39:41,100:102)
ans =
    14    15    14
    10    16    10
    10    10    10
```

The `colormap` function generates by default a 64-by-3 matrix with elements in the range 0–1. The values in the three columns represent the intensities of the red, green and blue (RGB) video components respectively. Each row of this matrix therefore defines a particular color by specifying its RGB components. The `image` function maps each element of its argument to a row in the colormap to find the color of that element. For example, `X(40,101)` has the value 16. Row 16 of the colormap has the three values

```
0.6784  0.3216  0.1922
```

(reddish) as you can easily verify with the statements

```
cm = colormap(map);
cm(16,:)
```

(The `map` colormap is also loaded from `earth`.) These values of RGB specify the color of the pixel 40 from the top and 101 from the left in the figure. Incidentally, you can use the statement

```
[xp yp] = ginput
```

to get the coordinates of a point in the figure (cross-hairs appears in the figure; click on the point whose coordinates you want). Clicking on the point in north-east Africa results in

```
xp =
101.8289  % from the left (column of X)
yp =
40.7032  % from the top (row of X)
```

Note that `xp` and `yp` correspond to the *columns* and *rows* of `X` respectively (imagine the matrix X superimposed on the figure).

There are a number of functions provided by MATLAB which generate colormaps, e.g. `jet` (the default), `bone`, `flag`, `prism`. See `help graph3d` for a complete list.

You can sample the various colormaps quite nicely with the following statement, which shows 64 vertical strips, each in a different color:

```
image(1:64),colormap(prism)
```

Or why not generate some random colors?

```
randmap(:,1) = rand(64,1);
randmap(:,2) = rand(64,1);
randmap(:,3) = rand(64,1);
image(1:64);colormap(randmap)
```

The function `colorbar` displays the current colormap vertically or horizontally in the figure with your graph, indicating how the 64 colors are mapped. Try it with the image of the Earth.

Note that 64 is the default length of a colormap. The functions that generate colormaps have an optional parameter specifying the length of the colormap.

12.4.2 Color of surface plots

When you draw a surface plot with a single matrix argument, e.g. `surf(z)`, the argument `z` specifies both the height of the surface and the color. As an example use the function `peaks` which generates a surface with a couple of peaks and valleys:

```
z = peaks;
surf(z), colormap(jet), colorbar
```

The colorbar indicates that the minimum element of `z` (somewhat less than −6) is mapped to row 1 of the colormap (R = 0, G = 0, B = 0.5625), whereas the maximum element (about 8) is mapped to row 64 (R = 0.5625, G = 0, B = 0).

You can specify the color with a second argument the same size as the first:

```
z = peaks(16);   % generates a 16-by-16 mesh
c = rand(16);
surf(z, c), colormap(prism)
```

Here the surface is tiled with a random pattern of 16-by-16 colors from the `prism` colormap.

In this form of `surf` each element of `c` is used to determine the color of the point in the corresponding element of `z`. By default MATLAB uses a process called *scaled mapping* to map from an element of `z` (or `c`) to the color in the colormap. The details of the scaling are determined by the `caxis` command. For further details see `help caxis` or the section **Coloring Mesh and Surface Plots** in **Using MATLAB: 3-D Visualization: Creating 3-D Graphs**.

You can exploit the facility to specify color in order to emphasize properties of the surface. The following example is given in the MATLAB documentation:

```
z = peaks(40);
c = del2(z);
surf(z, c)
colormap hot
```

The function `del2` computes the discrete Laplacian of a surface—the Laplacian is related to the curvature of the surface. Creating a color array from the Laplacian means that regions with similar curvature will be drawn in the same color. Compare the surface obtained this way with that produced by the statement

```
surf(P), colormap(hot)
```

In the second case regions with similar heights about the *x-y* plane have the same color.

Alternative forms of `surf` (and related surface functions) are:

```
surf(x, y, z)          % color determined by z
surf(x, y, z, c)       % color determined by c
```

12.4.3 Truecolor

The technique of coloring by means of a colormap is called *indexed* coloring—a surface is colored by assigning each data point an index (row) in the color map. The Truecolor technique colors a surface using explicitly specified RGB triplets. Here is another example from the MATLAB documentation (it also demonstrates the use of a multidimensional array):

```
z = peaks(25);
c(:,:,1) = rand(25);
c(:,:,2) = rand(25);
c(:,:,3) = rand(25);
surf(z, c)
```

The three 'pages' of `c` (indicated by its third subscript) specify the values of RGB respectively to be used to color the point in `z` whose subscripts are the same as the first two subscripts in `c`. For example, the RGB values for the color of the point `z(5,13)` are given by `c(5,13,1)`, `c(5,13,2)` and `c(5,13,3)` respectively.

12.5 Lighting and camera

MATLAB uses lighting to add realism to graphics, e.g. illuminating a surface by shining light on it from a certain angle. Here are two examples from the MATLAB documentation. Have a look at them.

```
z = peaks(25);
c(:,:,1) = rand(25);
c(:,:,2) = rand(25);
c(:,:,3) = rand(25);
surf(z,c,'facecolor','interp','facelighting','phong',...
          'edgecolor','none')
camlight right
```

The possibilities for the `facelighting` property of the surface object are `none`, `flat` (uniform color across each facet), `gouraud` or `phong`. The last two are the names of lighting algorithms. Phong lighting generally produces better results, but takes longer to render than Gouraud lighting. Remember that you can see all a surface object's properties by creating the object with a handle and using `get` on the object's handle.

This one's quite stunning:

```
[x y ] = meshgrid(-8 : 0.5 : 8);
r = sqrt(x.^2 + y.^2) + eps;
z = sin(r) ./ r;
surf(x,y,z,'facecolor','interp','edgecolor','none', ...
  'facelighting','phong')
colormap jet
daspect([5 5 1])
axis tight
view(-50, 30)
camlight left
```

For more information on lighting and the camera see the sections **Lighting as a Visualization Tool** and **Defining the View** in **Using MATLAB: 3-D Visualization**.

12.6 Saving, printing and exporting graphs

12.6.1 Saving and opening figure files

You can save a figure generated during a MATLAB session so that you can open it in a subsequent session. Such a file has the `.fig` extension.

- Select **Save** from the figure window **File** menu.
- Make sure the **Save as type** is `.fig`.

To open a figure file select **Open** from the **File** menu.

12.6.2 Printing a graph

You can print everything inside a figure window frame, including axis labels and annotations:

- To print a figure select **Print** from the figure window **File** menu.
- If you have a black and white printer, colored lines and text are 'dithered to gray' which may not print very clearly in some cases. In that case select **Page Setup** from the figure **File** menu. Select the **Lines and Text** tab and click on the **Black and white** option for **Convert solid colored lines to:**.

12.6.3 Exporting a graph

A figure may be exported in one of a number of graphics formats if you want to import it into another application, such as a text processor. You can also export it to the Windows clipboard and paste it from there into an application.

To export to the clipboard:

- Select **Copy Figure** from the figure window's **Edit** menu (this action copies to the clipboard).
- Before you copy to the clipboard you may need to adjust the figure's settings. You can do this by selecting **Preferences** from the figure's **File** menu. This action opens the **Preferences** panel, from which you can select **Figure Copy Template Preferences** and **Copy Options Preferences** to adjust the figure's settings.

  You may also have to adjust further settings with **Page Setup** from the figure's **File** menu.

To export a figure to a file in a specific graphics format:

- Select **Export** from the figure's **File** menu. This action invokes the **Export** dialogue box.
- Select a graphics format from the **Save as type** list, e.g. EMF (enhanced metafiles), JPEG, etc. You may need to experiment to find the best format for the target application you have in mind.

  For example, to insert a figure into a Word document, I find it much easier first to save it in EMF or JPEG format and then to insert the graphics file into the Word document, rather than to go the clipboard route (there are more settings to adjust that way).

For further details consult the section **Basic Printing and Exporting** in **Using MATLAB: Graphics**.

Summary

- MATLAB graphics objects are arranged in a parent–child hierarchy.
- A handle may be attached to a graphics object at creation; the handle may be used to manipulate the graphics object.
- If `h` is the handle of a graphics object, `get(h)` returns all the current values of the object's properties. `set(h)` shows you all the possible values of the properties.
- The functions `gcf`, `gca` and `gco` return the handles of various graphics objects.
- Use `set` to change the properties of a graphics object.
- A graph may be edited to a limited extent in plot edit mode, selected from the figure window (**Tools** -> **Edit Plot**). More general editing can be done with the Property Editor (**Edit** -> **Figure Properties** from the figure window).
- The most versatile way to create animations is by using the Handle Graphics facilities. Other techniques involve comet plots and movies.
- Indexed coloring may be done with colormaps.
- Graphs saved to `.fig` files may be opened in subsequent MATLAB sessions.
- Graphs may be exported to the Windows clipboard, or to files in a variety of graphics formats.

13

*Graphical User Interfaces (GUIs)

The objectives of this chapter are to introduce you to the basic concepts involved in writing your own graphical user interfaces, including:

- **Figure files.**
- **Application M-files.**
- **Callback functions.**

The script files we have written so far in this book have had no interaction with the user, except by means of the occasional `input` statement. Modern users, however, have grown accustomed to more sophisticated interaction with programs, by means of windows containing menus, buttons, drop-down lists, etc. This way of interacting with a program is called a *graphical user interface*, or GUI for short (pronounced 'gooey'), as opposed to a *text* user interface by means of command-line based `input` statements.

The MATLAB online documentation has a detailed description of how to write GUIs in MATLAB, to be found in **Using MATLAB: Creating Graphical User Interfaces**. This chapter presents a few examples illustrating the basic concepts.

Hahn, the original author (of the first two editions of this book) stated that he "spent a day or two reading MATLAB's documentation in order to write the GUIs in this chapter." He pointed out that he seldom had such fun programming!

13.1 Basic structure of a GUI

- MATLAB has a facility called GUIDE (Graphical User Interface Development Environment) for creating and implementing GUIs. You start it from the command line with the command `guide`.

- The `guide` command opens an untitled figure which contains all the GUI tools needed to create and lay out the GUI components (e.g. pushbuttons, menus, radio buttons, popup menus, axes, etc.). These components are called *uicontrols* in MATLAB (for user interface controls).
- GUIDE generates two files that save and launch the GUI: a *FIG-file* and an *application M-file*.
- The FIG-file contains a complete description of the GUI figure, i.e. whatever uicontrols and axes are needed for your graphical interface.
- The application M-file (the file you run to launch the GUI) contains the functions required to launch and control the GUI. The application M-file does not contain the code that lays out the GUI components; this information is saved in the FIG-file.

  The GUI is controlled largely by means of *callback functions*, which are subfunctions in the application M-file.
- Callback functions are written by you and determine what action is taken when you interact with a GUI component, e.g. click a button. GUIDE generates a callback function prototype for each GUI component you create. You then use the Editor to fill in the details.

13.2 A first example: getting the time

For our first example we will create a GUI with a single pushbutton. When you click the button the current time should appear on the button (Figure 13.1). Go through the following steps:

- Run `guide` from the command line. This action opens the Layout Editor (Figure 13.2).
- We want to place a pushbutton in the layout area. Select the pushbutton icon in the component palette to the left of the layout area. The cursor changes to a cross. You can use the cross either to select the position of the top-left corner of the button (in which it has the default size), or you can set the size of the button by clicking and dragging the cursor to the bottom-right corner of the button.

  A button imaginatively names `Push Button` appears in the layout area.
- Select the pushbutton (click on it). Right-click for its context menu and select **Inspect Properties**. Use the Property Inspector to change the button's `String` property to `Time`. The text on the button in the layout area will change to `Time`.

Figure 13.1 Click the button to see the time

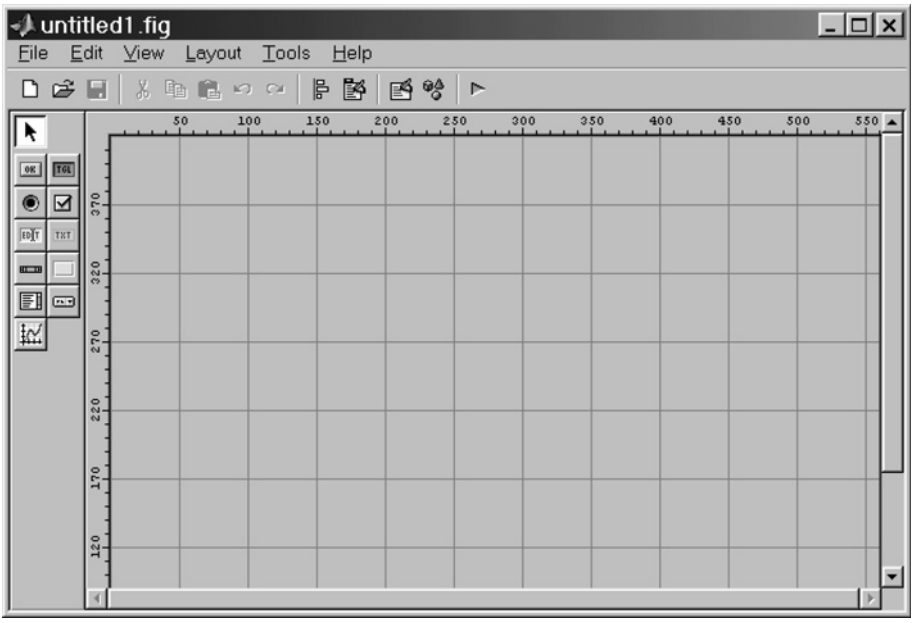


Figure 13.2 GUIDE Layout Editor

Note that the `Tag` property of the button is set to `pushbutton1`. This is the name GUIDE will use for the button's callback function when it generates the application M-file in a few minutes. If you want to use a more meaningful name when creating a GUI which may have many such buttons, now is the time to change the tag.

If you change a component's tag after generating the application M-file you have to change its callback function name by hand (and all references to the callback). You also have to change the name of the callback function in the component's `Callback` property. For example, suppose you have generated an application M-file with the name `myapp.m`, with a pushbutton having the tag `pushbutton1`. The pushbutton's `Callback` property will be something like

```
myapp('mybutton_Callback',gcbo,[],guidata(gcbo))
```

If you want to change the tag of the pushbutton to `Open`, say, you will need to change its `Callback` property to

```
myapp('Open_Callback',gcbo,[],guidata(gcbo))
```

➤ Back in the Layout Editor select your button again and right-click for the context menu. Select **Edit Callback**.

A dialog box appears for you to save your FIG-file. Choose a directory and give a filename of `time.fig`—make sure that the `.fig` extension is used.

GUIDE now generates the application M-file (`time.m`), which the Editor opens for you at the place where you now have to insert the code for whatever action you want when the `Time` button is clicked:

```
function varargout = pushbutton1_Callback(h,
                     eventdata, ... handles, varargin)
% Stub for Callback of the uicontrol handles.
                                          pushbutton1.
disp('pushbutton1 Callback not implemented yet.')
```

Note that the callback's name is `pushbutton1_Callback` where `pushbutton1` is the button's Tag property.

Note also that the statement

```
disp('pushbutton1 Callback not implemented yet.')
```

is automatically generated for you.

➤ If you are impatient to see what your GUI looks like at this stage click the **Activate Figure** button at the right-hand end of the Layout Editor toolbar.

An untitled window should appear with a `Time` button in it. Click the `Time` button—the message

```
pushbutton1 Callback not implemented yet
```

should appear in the Command Window.

- Now go back to the (M-file) Editor to change the `pushbutton1_Callback` function. Insert the following lines of code, which are explained below:

```
% get time
d = clock;
% convert time to string
time = sprintf('%02.0f:%02.0f:%02.0f',d(4),d(5),
                  d(6));
% change the String property of pushbutton1
set(gcbo,'String',time)
```

The function `clock` returns the date and time as a six-component vector in the format

```
[year month day hour minute seconds]
```

The `sprintf` statement converts the time into a string `time` in the usual format, e.g. `15:05:59`

- The last statement is the most important one as it is the basis of the button's action:

```
set(gcbo,'String',time)
```

It is the button's `String` property which is displayed on the button in the GUI. It is therefore this property which we want to change to the current time when it is clicked.

The command `gcbo` stands for 'get callback object', and returns the handle of the current callback object, i.e. the most recently activated uicontrol. In this case the current callback object is the button itself (it could have been some other GUI component). The `set` statement then changes the `String` property of the callback object to the string `time` which we have created.

Easy!

- Save the application M-file and return to the Layout Editor. For completeness give the entire figure a name. Do this by right-clicking anywhere in the layout area outside the `Time` button. The context menu appears. Select **Property Inspector**. This time the Property Inspector is looking at the

figure's properties (instead of the pushbutton's properties) as indicated at the top of its window. Change the `Name` property to something suitable (like `Time`).

- Finally, click **Activate Figure** again in the Layout Editor. Your GUI should now appear in a window entitled `Time`. When you click the button the current time should appear on the button.
- If you want to run your GUI from the command line, enter the application M-file name.

13.2.1 Exercise

You may have noticed that graphics objects have a `Visible` property. See if you can write a GUI with a single button which disappears when you click on it! This exercise reminds me of a project my engineering friends had when I was a first-year student. They had to design a black box with a switch on the outside. When it was switched on a lid opened, and a hand came out of the box to switch if off …

13.3 Newton again

Our next example is a GUI version of Newton's method to find a square root: you enter the number to be square-rooted, press the **Start** button, and some iterations appear on the GUI (Figure 13.3).

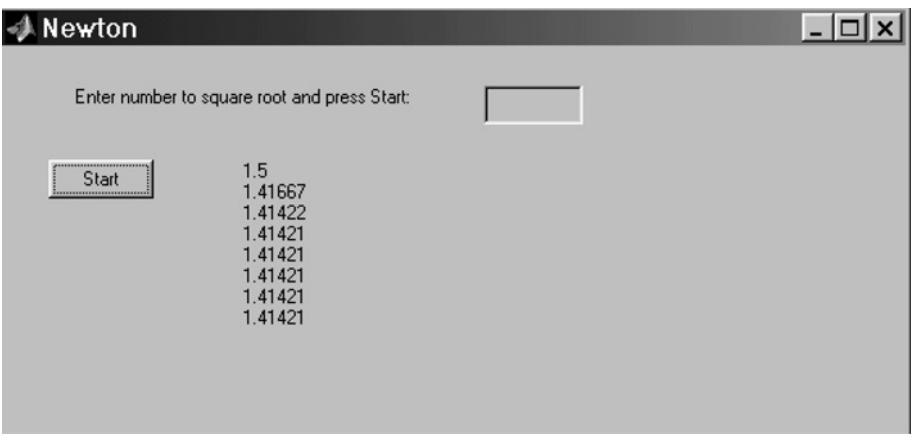


Figure 13.3 Square roots with Newton's method.

We will use the following uicontrols from the component palette in the Layout Editor:

- a static text control to display the instruction `Enter number to square root and press Start:` (static text is not interactive, so no callback function is generated for it);
- an edit text control for the user to enter the number to be square-rooted;
- a pushbutton to start Newton's method;
- a static text control for the output, which will consist of several iterations.

  You can of course send the output to the command line with `disp`.
- We will also make the GUI window resizable (did you notice that you couldn't resize the `Time` window in the previous example?).

Proceed as follows:

- Run `guide` from the command line. When the Layout Editor starts right-click anywhere in the layout area to get the figure's Property Inspector and set the figure's `Name` property to `Newton`.
- Now let's first make the GUI resizable. Select **Tools** in the Layout Editor's toolbar and then **Application Options**. There are three **Resize behavior** options: **Non-resizable** (the default), **Proportional** and **User-specified**. Select **Proportional**. Note that under **Proportional** when you resize the window the GUI components are also resized, although the font size of labels does not resize.
- Place a static text control for the instruction to the user in the layout area. Note that you can use the Alignment Tool in the Layout Editor toolbar to align controls accurately if you wish. You can also use the **Snap to Grid** option in the **Layout** menu. If **Snap to Grid** is checked then any object which is moved to within nine pixels of a grid line jumps to that line. You can change the grid spacing with **Layout -> Grid and Rulers -> Grid size**.

  Use the Property Inspector to change the static text control's `String` property to `Enter number to square root and press Start:`
- Place an edit text control on the layout area for the user to input the number to be square-rooted. Set its `String` property to blank. Note that you don't need to activate the Property Inspector for each control. Once the Property Inspector has been started it switches to whichever object you select in the layout area. The tag of this component will be `edit1` since no other edit text components have been created.

- Activate the figure at this stage (use the Activate Figure button in the Layout Editor toolbar), and save it under the name `Newton.fig`. The application M-file should open in the Editor. Go to the `edit1_Callback` subfunction and remove the statement

  ```
  disp('edit1 Callback not implemented yet.')
  ```

  No further coding is required in this subfunction since action is to be initiated by the `Start` button.

  Remember to remove this `disp` statement from each callback subfunction as you work through the code.

- Insert another static text control in the layout area for the output of Newton's method. Make it long enough for five or six iterations. Set its `String` property to `Output` and its `HorizontalAlignment` to `left` (so that output will be left-justified).

  Make a note of its tag, which should be `text2` (`text1` being the tag of the other static text control).

- Place a pushbutton in the layout area. Set its `String` property to `Start`. Its tag should be `pushbutton1`.

- Activate the figure again so that the application M-file gets updated. We now have to program the `pushbutton1_Callback` subfunction to

  – get the number to be square-rooted from the edit text control;

  – run Newton's method to estimate the square root of this number;

  – put the output of several iterations of Newton's method in the `Output` static text control;

  – clear the edit text control in readiness for the next number to be entered by the user.

  These actions are achieved by the following code, which is explained below:

```
function varargout = pushbutton1_Callback(h,
                     eventdata, ... handles, varargin)
% Stub for Callback of the uicontrol handles.
                                          pushbutton1.

a = str2num(get(handles.edit1,'String'));
x = a; % initial guess

for i = 1:8
  x = (x + a/x)/2;
  str(i) = sprintf('%g',x);
end
```

```
set(handles.edit1,'String','');
set(handles.text2,'String',str);
```

If you have read the helpful comments generated in the application M-file you will have seen that the input argument `handles` is a structure containing handles of components in the GUI, using their tags as fieldnames. This mechanism enables the various components to communicate with each other in the callback subfunctions. This is why it is not a good idea to change a tag after its callback subfunction has been generated; you would then need to change the name of the callback function as well as any references to its handle.

When you type anything into an edit text control, its `String` property is automatically set to whatever you type. The first statement

```
a = str2num(get(handles.edit1,'String'));
```

uses the handle of the edit text control to get its `String` property (assuming it to have been correctly entered—note that we are not trying to catch any user errors here). The `str2num` function converts this string to the number `a`.

A `for` loop then goes through eight iterations of Newton's method to estimate the square root of `a`.

The variable `x` in the loop contains the current estimate. `sprintf` converts the value of `x` to a string using the `g` (general) format code. The resulting string is stored in the `i`th element of a cell array `str`. Note the use of curly braces to construct the cell array. This use of a cell array is the easiest way to put multiple lines of text into an array for sending to the `text2` static text.

The statement

```
set(handles.edit1,'String','');
```

clears the display in the edit text control.

Finally, the statement

```
set(handles.text2,'String',str);
```

sets the `String` property of the `text2` static text control to the cell array containing the output.

➤ Save your work and activate the figure again to test it.

13.4 Axes on a GUI

This example demonstrates how to plot graphs in axes on a GUI. We want a button to draw the graph, one to toggle the grid on and off, and one to clear the axes (Figure 13.4). The only new feature in this example is how to plot in axes which are part of the GUI. We address this issue first.

➤ Open a new figure in the GUIDE Layout Editor. Right-click anywhere in the layout area to open the figure's Property Inspector. Note that the figure has a `HandleVisibility` property which is set to `off` by default. This prevents any drawing in the figure. For example, you would probably not want plot statements issued at the command-line to be drawn on the GUI. However, you can't at the moment draw on the GUI from within the GUI either! The answer is to set the figure's `HandleVisibility` property to `callback`. This allows any GUI component to plot graphs on axes which are part of the GUI, while still protecting the GUI from being drawn all over from the command line.

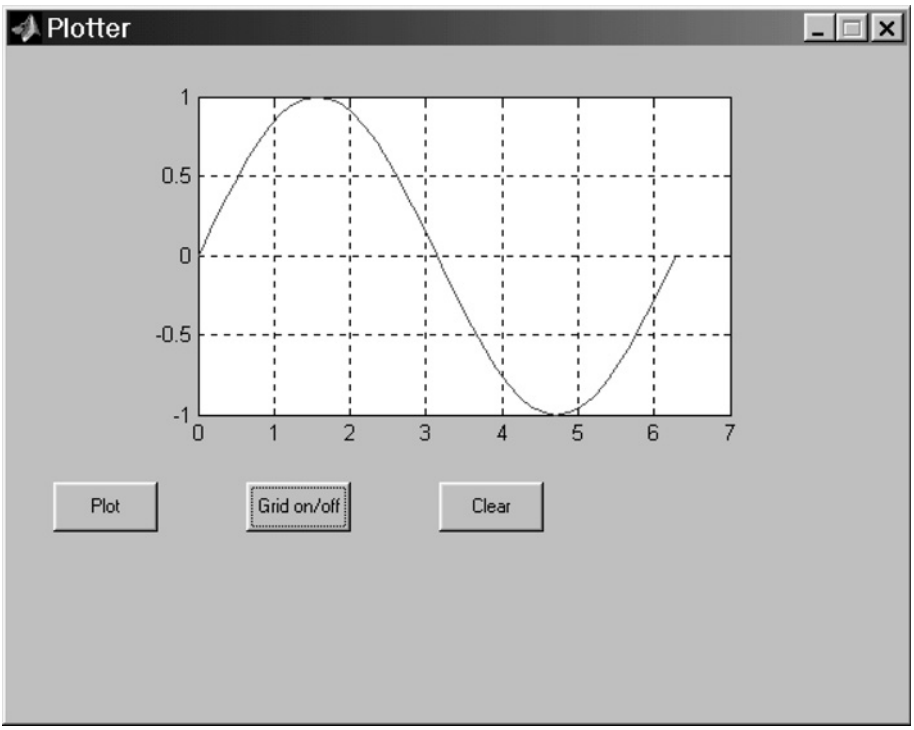


Figure 13.4 Axes on a GUI.

- Place an axes component on the GUI. Its `HandleVisibility` should be `on`.
- Place a pushbutton (tag `pushbutton1`) on the GUI and change its `String` property to `Plot`. Save the figure (if you have not already) under the name `Plotter` and generate the application M-file. Edit the pushbutton's callback subfunction to draw any graph, e.g. `sin(x)`.
- Place `pushbutton2` on the GUI to toggle the grid and put the command `grid` in its callback subfunction.
- Put `pushbutton3` on the GUI with the command `cla` to clear the axes.
- Save and reactivate the figure. Test it out. Change the figure's `HandleVisibility` back to `off`, activate the figure, and see what happens.

  Note that the figure object has a `IntegerHandle` property set to `off`. This means that even if `HandleVisibility` is `on` you can't plot on the axes from the command line (or another GUI) because a figure needs an integer handle in order to be accessed. If you want to be able to draw all over your GUI from the command line just set the figure's `IntegerHandle` to `on`.

13.5 Adding color to a button

The final example in this chapter shows you how to add an image to a button.

- Create a GUI with a single pushbutton in it.
- Set the button's `Units` property to `pixels`.
- Expand the button's `Position` property and set `width` to 100 and `height` to 50.
- Insert the following code in the button's callback subfunction:

```
a(:,:,1) = rand*ones(50,100);
a(:,:,2) = rand*ones(50,100);
a(:,:,3) = rand*ones(50,100);
set(h,'CData',a);
```

- The matrix `a` defines a truecolor image the same size as the pushbutton, with each pixel in the image the same (random) color.

  The third dimension of `a` sets the RGB video intensities. If, for example, you would prefer the button to be a random shade of green, say, try

```
a(:,:,1) = zeros(50,100);
a(:,:,2) = rand*ones(50,100);
a(:,:,3) = zeros(50,100);
```

- The input argument `h` of the callback subfunction is the callback object's handle (obtained from `gcbo`).
- An image may also be added to a toggle button control.

Summary

- GUIs are designed and implemented by MATLAB's GUIDE facility.
- The `guide` command starts the Layout Editor, which is used to design GUIs.
- GUIDE generates a FIG-file, which contains the GUI design, and an application M-file, which contains the code necessary to launch and control the GUI.
- You launch the GUI by running its application M-file.
- Many GUI components have callback subfunctions in the application M-file where you write the code to specify the components' actions.
- You use the Property Inspector to set a GUI component's properties at the design stage.
- The handles of all GUI components are fields in the `handles` structure in the application M-file. These handles are available to all callback subfunctions and enable you to find and change components' properties.

This page intentionally left blank

Part II

Applications

Part II is on applications. Since this is an introductory course, the applications are not extensive. They are illustrative. You ought to recognize that the kinds of problems you actually can solve with MATLAB are much more challenging than the examples provided. More is said on this matter at the beginning of Chapter 14. The goal of this part is to scratch the surface of the true power of tools like MATLAB and to show how you can use it in a variety of ways.

This page intentionally left blank

14

Dynamical systems

The objective of this chapter is to discuss the importance of learning to use tools like MATLAB. MATLAB and companion toolboxes provide engineers, scientists, mathematicians, and students of these fields with an environment for technical computing applications. It is much more than a programming language like C, C++ or Fortran. Technical computing includes mathematical computation, analysis, visualization, and algorithm development. The MathWork website describes 'The Power of Technical Computing with MATLAB' as follows:

- **Whatever the objective of your work—an algorithm, an analysis, a graph, a report, or a software simulation—technical computing with MATLAB lets you work more effectively and efficiently. The flexible MATLAB environment lets you perform advanced analyses, visualize data, and develop algorithms by applying the wealth of functionality available. With its more than 1000 mathematical, statistical, scientific and engineering functions, MATLAB gives you immediate access to high-performance numerical computing. This functionality is extended with interactive graphical capabilities for creating plots, images, surfaces, and volumetric representations.**
- **The advanced toolbox algorithms enhance MATLAB's functionality in domains such as signal and image processing, data analysis and statistics, mathematical modeling, and control design. Toolboxes are collections of algorithms, written by experts in their fields, that provide application-specific numerical, analysis, and graphical capabilities. By relying on the work of these experts, you can compare and apply a number of techniques without writing code. As with MATLAB algorithms, you can customize and optimize toolbox functions for your project requirements.**

In this chapter we are going to examine the application of MATLAB capabilities to four relatively simple problems in engineering science. The problems are the deflection of a cantilever beam subject to a uniform load, a single-loop closed electrical circuit, the free fall problem, and an extension of the projectile problem discussed in Chapter 3. The first problem is on a structural element you will investigate in your first course in engineering mechanics. The structural

element is the cantilever beam, which is one of the primary elements of engineered structures, e.g. buildings and bridges. We examine the deflection of this beam with a constant cross-section when subject to a uniform distribution of load, e.g. its own weight.

The second problem is an examination of the equation that describes the 'flow' of electrical current in a simple closed-loop electrical circuit. You examine this type of problem in your first course in electrical science.

The third problem is the free fall of an object in a gravitational field with constant acceleration, g. This is one of the first problems you examine in your first course in physics. We examine the effect of friction on the free-fall problem and, hence, learn that with friction (i.e. air resistance) the object can reach a terminal velocity.

The fourth problem is an extension of the projectile that takes into account air resistance (you learn about air resistance in your first course in fluid mechanics). By an examination of this problem, you will learn why golfers do not hit the ball from the tee at 45 degrees from the horizontal (which is the optimum angle for the furthest distance of travel of a projectile launched in frictionless air).

Before we begin, you must keep in mind that this is *not* a book on engineering or science; it is an introduction to the essential elements of MATLAB, a powerful technical analysis tool. In this and subsequent chapters you will scratch the surface of its power by solving some relatively simple problems that you have encountered or will encounter early in your science or engineering education. Keep in mind that MATLAB is a tool that can be used for a number of very productive purposes. One purpose is to learn how to write programs to investigate topics you confront in the classroom. You can extend your knowledge of MATLAB by self-learning to apply the utilities and any of the toolboxes that are available to you to solve technical problems. This tool requires a commitment from you to continue to discover and to use the power of the tool as part of your continuing education. Developing high-quality computer programs within the MATLAB environment in order to solve technical problems requires that you use the utilities available. If you need to write your own code you should, if at all possible, include the utilization of generic solvers of particular types of equations that are available in MATLAB. Of course, to do this you need to learn more advanced mathematics, science and engineering. You will certainly have the opportunity to do this after freshman year.

As a student majoring in science and engineering you will take at least five semesters of mathematics. The fifth course is usually titled Advanced Calculus, Advanced Engineering Mathematics, Applied Mathematics, or Methods of Mathematical Physics. Topics covered in this course are typically ordinary differential

equations, systems of differential equations (including nonlinear systems), linear algebra (including matrix and vector algebra), vector calculus, Fourier analysis, Laplace transforms, partial differential equations, complex analysis, numerical methods, optimization, probability and statistics. By clicking the question mark (?) near the center of the uppermost toolbar in the MATLAB desktop, as you already know, opens the help browser. Scan the topics and find utilitites and examples of approaches to solve problems in all of these subjects and more.

The Control Systems Toolbox is an example of one of the numerous engineering and scientific toolboxes that have been developed to advance the art of doing science and engineering. SIMULINK and Symbolics are also very important toolboxes. The latter two are included with your 'student version' of MATLAB. One of the big advantages of SIMULINK, in addition to being a graphical algorithm-development and programming environment, is that it is useful when using the computer to measure, record and access data from laboratory experiments or data from industrial-process monitoring systems. Symbolics is very useful because it allows you to do symbolic mathematics, e.g. integrating functions like the ones you integrate in your study of the calculus. All of the mathematical subjects investigated by you, as a science or engineering major, are applied in your science and engineering courses. They are the same techniques that you will apply as a professional (directly or indirectly depending on the software tools available within the organization that you are employed). Thus, you can use your own copy of MATLAB to help you understand the mathematical and software tools that you are expected to apply in the classroom as well as on the job.

14.1 Cantilever beam

In this section we want to examine the problem of the cantilever beam. The beam and its deflection under load is illustrated in Figure 14.1 below, which is generated by the script *m-file* created to solve the problem posed next. Many structural mechanics formulae are available in *Formulas for Stress and Strain*, Fifth Edition by Raymond J. Roark and Warren C. Young, published by McGraw-Hill Book Company (1982). For a uniformly loaded span of a cantilever beam attached to a wall at $x = 0$ with the free end at $x = L$, the formula for the vertical displacement from $y = 0$ under the loaded condition with y the coordinate in the direction opposite that of the load can be written as follows:

$$Y = \frac{y\,24EI}{wL^4} = -\left(X^4 - 4X^3 + 6X^2\right),$$

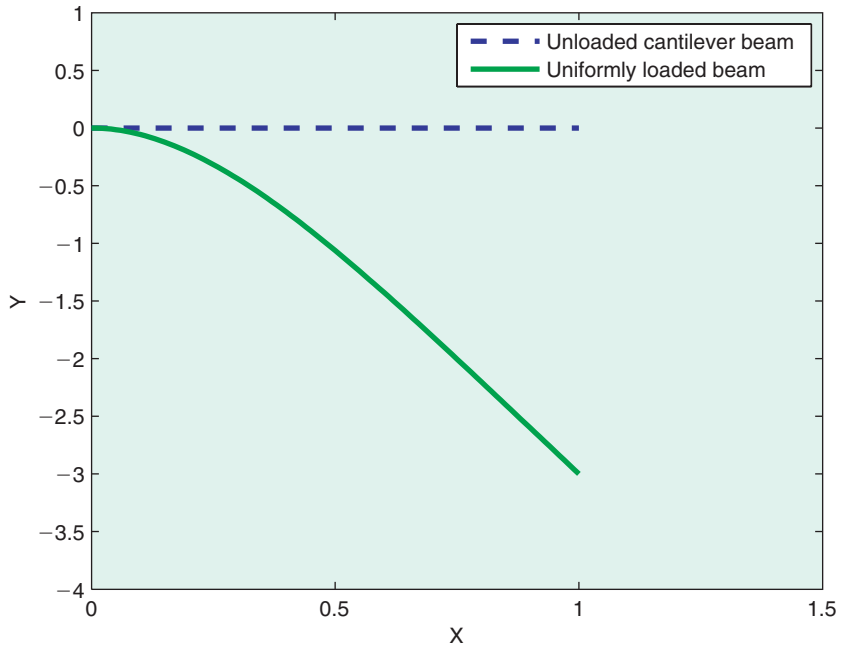


Figure 14.1 Deflection of a cantilever beam. The vertical deflection of a uniformly loaded cantilever beam

where $X = x/L$, E is a material property known as the modulus of elasticity, I is a geometric property of the cross-section of the beam known as the moment of inertia, L is the length of the beam extending from the wall from which it is mounted, and w is the load per unit width of the beam (this is a two-dimensional analysis). The formula was put into dimensionless form to answer the following question: What is the shape of the deflection curve when the beam is in its loaded condition and how does it compare with its unloaded perfectly horizontal orientation? The answer will be provided graphically. This is easily solved by using the following script:

```
%
% The deflection of a cantilever beam under a uniform
    %load
% Script by D.T.V. ........ September 2006
%
% Step 1: Select the distribution of X's from 0 to 1
% where the deflections to be plotted are to be
    %determined.
%
X = 0:.01:1;
%
```

```
% Step 2: Compute the deflections Y at eaxh X. Note that
% YE is the unloaded position of all points on the beam.
%
Y = - ( X.^4 - 4 * X.^3 + 6 * X.^2 );
YE = 0;
%
% Step 3: Plot the results to illustrate the shape of the
% deflected beam.
%
plot([0 1],[0 0],'--',X,Y,'LineWidth',2)
axis([0,1.5,-4, 1]),title('Deflection of a cantilever ...
  beam')
xlabel('X'),ylabel('Y')
legend('Unloaded cantilever beam','Uniformly loaded ...
  beam')
%
% Step 4: Stop
```

Again, the results are plotted in Figure 14.1. It looks like the beam is deflected tremendously. The actual deflection will not necessarily be so dramatic once the actual material properties are substituted to determine y and x in, say, meters. The scaling, i.e. rewriting the equation in dimensionless form in terms of the unitless quantities Y and X, provides insight into the shape of the curve. In addition, the shape is independent of the material as long as it is a material with uniform properties and the geometry has a uniform cross-sectional area, e.g. a rectangular area of height h and width b that is independent of the distance along the span (or length) L.

14.2 Electric current

When you investigate electrical science you investigate circuits with a variety of components. In this section we will solve the governing equation to examine the dynamics of a single closed-loop electrical circuit. The loop contains a voltage source, V (e.g. a battery), a resistor, R (i.e. an energy dissipation device), an inductor, L (i.e. an energy storage device), and a switch which is instantaneously closed at time $t = 0$. From Kirchhoff's law, as described in the book by Ralph J. Smith entitled *Circuits, Devices, and Systems*, published by John Wiley & Sons, Inc. (1967), the equation describing the response of this system from an initial state of zero current is as follows:

$$L\frac{di}{dt} + Ri = V,$$

where i is the current. At $t = 0$ the switch is engaged to close the circuit and initiate the current. At this instant of time the voltage is applied to the resistor

and inductor (which are connected in series) instantaneously. The equation describes the value of i as a function of time after the switch is engaged. Hence, for the present purpose, we want to solve this equation to determine i versus t graphically. Rearranging this equation, we get

$$\frac{di}{dt} + \frac{R}{L} i = \frac{V}{L}.$$

The solution, by inspection (another method that you learn when you take differential equations), is

$$i = \frac{V}{R}\left(1 - e^{-\frac{R}{L}t}\right).$$

This solution is checked with the following script:

```
%
% Script to check the solution to the soverning
% equation for a simple circuit, i.e. to check
% that
%       i = (V/R) * (1 -  exp(-R*t/L))
%
% is the solution to the following ODE
%
%       di/dt + (R/L) * i - V/L = 0
%
% Step 1: We will use the Symbolics tools; hence,
% define the symbols as follows
%
        syms i V R L t
%
% Step 2: Construct the solution for i
%
        i = (V/R) * ( 1 - exp(-R*t/L) );
%
% Step 3: Find the derivative of i
%
       didt = diff(i,t);
%
% Step 4: Sum the terms in ODE
%
       didt + (R/L) * i - V/L;
%
% Step 5: Is the answer ZERO?
%
      simple(ans)
%
% Step 6: What is i at t = 0?
```

```
%
subs(i,t,0)
%
% REMARK: Both answers are zero; hence,
%         the solution is correct and the
%         initial condition is correct.
%
% Step 7: To illustrate the behavior of the
%         current, plot i vs. t for V/R = 1
%         and R/L = 1. The curve illustrates
%         the fact that the current approaches
%         i = V/R exponentially.
%
       V = 1; R = 1; L = 1;
       t = 0 : 0.01 : 6;
       i = (V/R) * ( 1 - exp(-R.*t/L) );
       plot(t,i,'ro'), title(Circuit problem example)
xlabel('time, t'),ylabel('current, i')
%
```

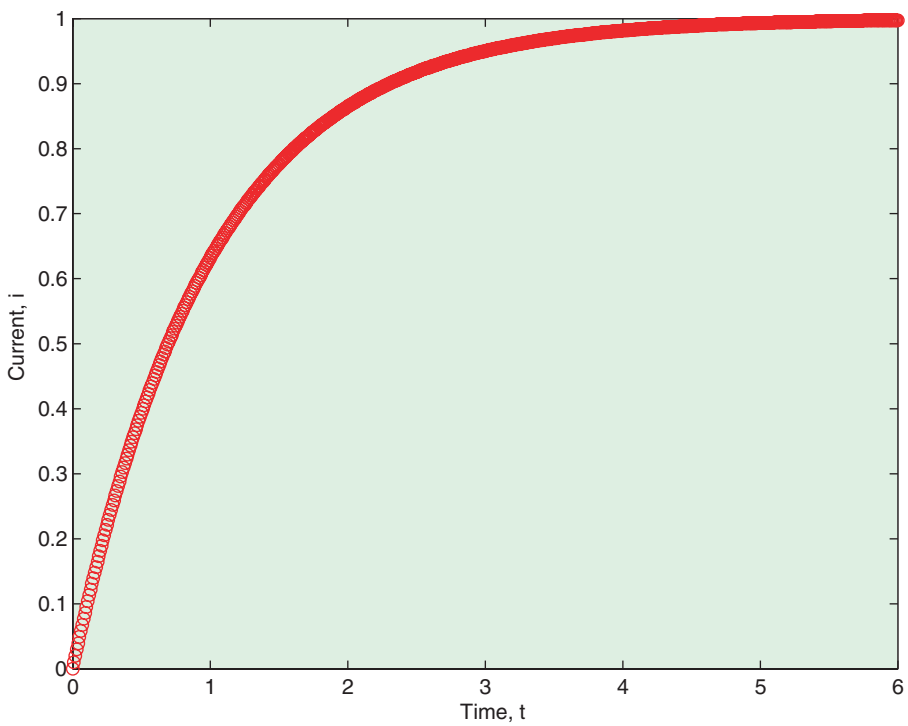


Figure 14.2 Example of a circuit problem. Exponential approach to steady current condition of a simple *RL* circuit with an instantaneously applied constant voltage

When you run this script you will prove that the solution given above is correct. The figure that illustrates the solution to the problem posed is reproduced in Figure 14.2. This concludes this exercise related to electrical circuit theory that you will learn more about in your course on electrical science.

14.3 Free fall

In this section we are going to use MATLAB not only to investigate the problem of free fall with friction (or air resistance), but also to use MATLAB to check the theoretical results found in the literature. This example deals with the application of MATLAB for computing and graphically presenting information for analysis of the problem of free fall. This investigation is to examine the effect of air resistance on the distance of free fall in 5 seconds from a location $y = 0$, where the object is initially at rest (y is in the direction of gravity). Hence, we want to determine the distance $y = L$ that an object falls from a state of rest with and without air resistance. In the book entitled *Introduction to Theoretical Mechanics* by R. A. Becker, which was published by McGraw-Hill Book Company (1954), the equations for free fall are given. The three cases of interest are as follows:

1. Without air resistance:

$$a = \frac{d^2y}{dt^2} = g, \quad v = \frac{dy}{dt} = gt, \quad y = \frac{1}{2}gt^2,$$

where a is the acceleration of the object, v is its speed and y is the distance of its free fall from the start of motion at $t = 0$.

2. With resistance proportional to the linear power of velocity:

$$\begin{aligned} a &= \frac{d^2y}{dt^2} = g - k\frac{dy}{dt}, \quad v = \frac{dy}{dt} = \frac{g}{k}\left(1 - e^{-kt}\right), \\ y &= \frac{g}{k}t - \frac{g}{k^2}\left(1 - e^{-kt}\right). \end{aligned}$$

3. With resistance proportional to the second power of velocity:

$$\begin{aligned} a &= \frac{d^2y}{dt^2} = g - k\left(\frac{dy}{dt}\right)^2, \quad v = \frac{dy}{dt} = \frac{1}{2}\frac{g}{k}\tanh\left(\frac{gk}{2}t\right), \\ y &= \frac{1}{k}\log_e\left[\cosh\left(gkt/2\right)\right]. \end{aligned}$$

For all three cases the initial condition is $y = v = 0$ at $t = 0$. (You will learn more about ordinary differential equations in your third semester of mathematics. In addition, in a first course in fluid mechanics you will learn some of the details

about air resistance. In particular, for air resistance associated with 'laminar flow' problems, case 2 applies. In many practical situations it is case 3 that is usually of interest; this is the case where 'turbulent flow' is important.)

Let us consider the following problem: Let us assume the above equations are correct. Note that they are written for a unit of mass. Let us also assume $g = 9.81$ m/s^2 and $k = 0.2$ for the air-drag parameter. Finally, let us answer the following:

(a) What is y in meters for $t = 5$ seconds of free fall for the three cases.
(b) What are the terminal speeds in meters per second for cases 2 and 3 for the specified air-drag parameter?
(c) Is the terminal speed reached at $t = 5$ seconds?

Note:

The terminal speeds are g/k and $(g/k)/2$ for cases 2 and 3, respectively, where the terminal speed is the time independent or steady speed reached after a sufficient distance of free fall. It is the speed at which the gravitational force balances the air resistance force. From Part I, the Essentials, we learned that MATLAB is quite useful in this type of problem because it has the capability of computing the elementary functions in the formulae above.

As part of providing an answer to the questions raised, we want to examine the distance of free fall within the 5 seconds of flight time with the equations for y given above. Let us examine the influence of air resistance on free fall graphically; thus, we need to plot the distance y versus t from $t = 0$ to $t = 5$ seconds. We shall plot all three curves on one figure for direct comparison. We will show that for a short period of time (significantly less than the 5 seconds) after the onset of motion all three curves are on top of each other. We will also show that at $t = 5$ seconds the distances of free fall are quite different.

The steps in the structure plan and the MATLAB code are as follows:

```
%
%      Free fall analysis (saved as FFall.m):
%       Comparison of exact solutions of free
%       fall with zero, linear and quadratic
%       friction for t = 0 to 5 seconds.
%
% Script by D. T. V. ......... September 2006.
%
```

```
% Step 1: Specify constants
%
k = 0.2; % Friction coefficient provided in the problem
           statement.
g = 9.81; % Acceleration of gravity in m/s/s.
%
% Step 2: Selection of time steps for computing solutions
%
dt = .01;
%
% Step 3: Set initial condition (the same for all cases)
%
t(1) = 0.; v(1) = 0.; y(1) = 0.;
%
t = 0:dt:5;
%
% Step 4: Compute exact solutions at each time step from
          %t = 0 to 5.
%
%    (a) Without friction:
%
        v = g * t;
        y = g * t.^2 * 0.5;
%
%    (b) Linear friction
%
         velf = (g/k) * (1. - exp(-k*t));
      yelf = (g/k) * t - (g/(k^2)) * (1.-exp(-k*t));
%
%    (c) Quadratic friction
%
      veqf = sqrt(g/k) * tanh( sqrt(g*k) * t);
      yeqf = (1/k) * log(cosh( sqrt(g*k) * t) );
%
% Step 5: Computation of the terminal speeds (cases with
          %friction)
%
      velfT = g/k;
      veqfT = sqrt(g/k);
%
% Step 6: Graphical comparison
%
 plot(t,y,t,yelf,t,yeqf);title('Fig 1. Comparison of
   results')
 xlabel(' Time, t')
 ylabel(' Distance, y ')
 figure
```

```
 plot(t,v,t,velf,t,veqf);title('Fig. 2. Comparison of
   results')
 xlabel(' Time, t')
 ylabel(' Speed, v ')
%
% Step 7: Comparison of distance and speed at t = 5
%
disp(' ');
fprintf(' y(t) = %f, yelf(t) = %f, yeqf(t) = %f at t =
  %f ... \n',...
     y(501),yelf(501),yeqf(501),t(501))
disp(' ');
fprintf(' v(t) = %f, velf(t) = %f, veqf(t) = %f at
  t = %f ... \n',...
    y(501),yelf(501),yeqf(501),t(501))
%
% Step 8: Comparison of terminal velocity
%
disp(' ');
fprintf('  velfT = %f, veqfT = %f  ... \n',...
           velfT,veqfT)
%
% Step 9: Stop
%
```

The command window execution of the file above (named FFALL.m) gives the comparisons in Figures 14.3 and 14.4 and the printed results as follows:

```
>> FFall

  y(t) = 122.625000, yelf(t) = 90.222433, yeqf(t)
       = 31.552121 at t = 5.000000 ...
  v(t) = 49.050000, velf(t) = 31.005513, veqf(t)
       = 7.003559 at t = 5.000000 ...
  velfT = 49.050000, veqfT = 7.003571  ...
```

The figures illustrate, as we may have expected, that for no friction the object falls the furthest distance. The case with quadratic friction reaches terminal velocity well within the 5 seconds examined. The linear friction case did not reach terminal speed, yet it is moving at a slower velocity as compared with the no friction case. Keep in mind that a unit mass object is falling with a friction coefficient $k = 0.2$. The same k was used for both the second and third cases. Within the first half second from the time of release the three curves are not distinguishable illustrating the fact that it takes a little time before the friction effects are felt. At the time of 5 seconds after release, the speed and the distance fallen are quite different. It is not surprising that quadratic friction

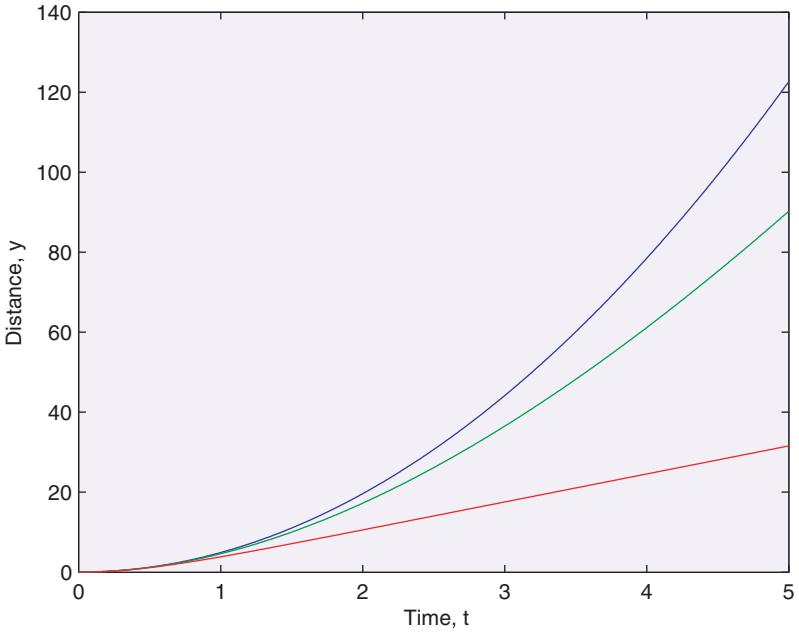


Figure 14.3 Comparison of free-fall distance: Top curve is for no friction, middle curve is for linear friction and the bottom curve is for quadratic friction

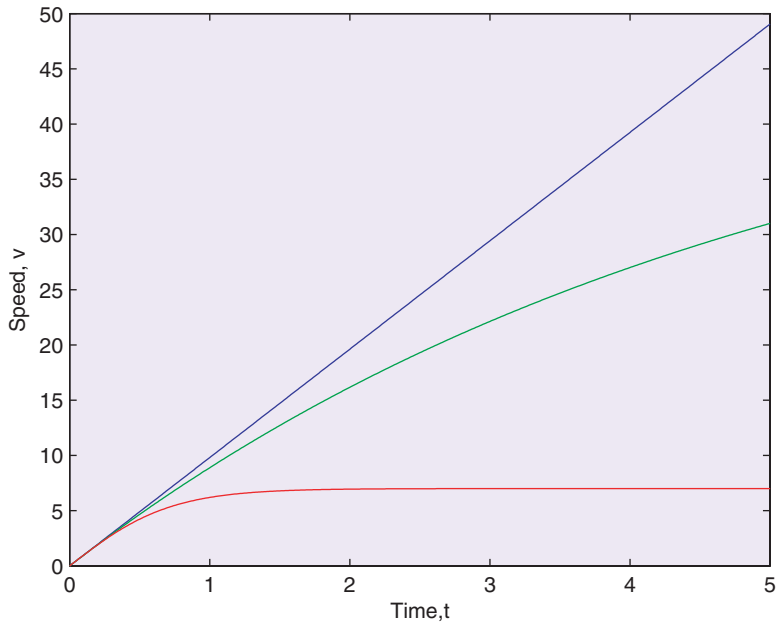


Figure 14.4 Comparison of free-fall speed: Top curve is for no friction, middle curve is for linear friction and the bottom curve is for quadratic friction

slows the object quicker because the air resistance (or friction) is proportional to the speed squared, which is significantly larger than speed to the first power (as it is in the linear-friction case).

The above analysis was based on the exact solutions of the free-fall problem. Let us now use the symbolic tools to check the theoretical results found in the literature. The case that we will examine is Case 2, the linear-friction case. The following script was implemented in the Command Window. The responses of MATLAB are also reproduced.

```
%
% The formula for the distance of free fall
% of an object from rest with linear friction
% is as follows:
%
% y = (g / k) * t - (g/k^2) * [ 1 - exp^(- k t)].
%
% To check the theory, we want to differentiate
% this twice to determine the formulae for velocity
% v and acceleration a, respectively. The results
% should match the published results.
%
% Step 1: Define the symblic variables
%
syms g k t y
%
% Step 2: Write the formula for y = f(t)
%
y = (g/k) * t - (g/k^2) * ( 1 - exp(-k * t));
%
% Step 3: Determine the velocity
%
v = diff(y,t);
%
% Step 4: Determine the acceleration
%
a = diff(v,t);
%
% Step 5: Print the v and a formulae and compare with
% the published results
%
v, a
%
% Step 6: Determine a from published formula and v.
%
a2 = g - k * v;
```

```
% Step 7: Simplify to somplest form
a2 = simple(a2)
%
% Step 6: Stop. REMARK: Results compare exactly. The
% results printed in the command window after executing
% this script are as follows:
%   v = g/k-g/k*exp(-k*t)
%   a  = g*exp(-k*t)
%   a2 = g/exp(k*t)
% These results verify the conclusion that the published
% formulae are correct.
%
```

Let us next consider an approximate method for solving the linear-friction case. This procedure is something you could have implemented if you didn't have the exact solution. More on numerical methods is provided in the last chapter of this book.

The equations for free-fall acceleration and velocity are differential equations. The following approximate analysis of a differential equation is called finite-differences. Let us consider the problem of free fall with linear friction (i.e. air drag that is linearly proportional to the speed of free fall). The formula

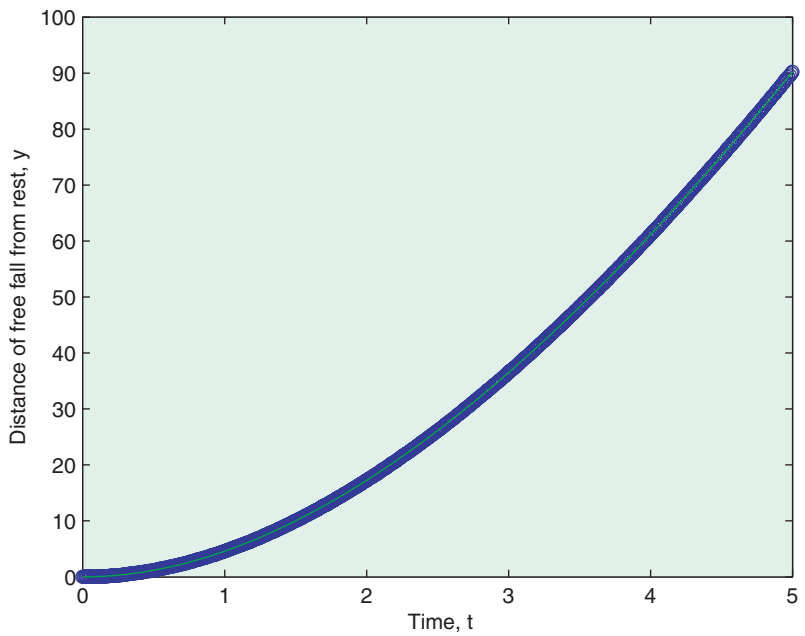


Figure 14.5 Comparison of free-fall distance. Speed: Exact is the line, circles are for the finite-difference approximate solution

(or equation) for this problem and its exact solution are given above. In the following analysis a script to solve this problem by the approximate method described below was written and executed for the same interval of time. This is described next. Then the approximate solution is compared with the exact solution graphically. For a unit mass, the formula that describes the velocity of free fall from rest with air resistance proportional to the linear power of velocity can be written as follows:

$$\frac{dv}{dt} = g - kv,$$

Let us approximate this equation by inverting the fundamental theorem of differential calculus, i.e. let us rewrite it in terms of the definition of what a derivative is before the appropriate limit is taken. What is meant by this is to write this equation for a small interval of time $\Delta t = t(n+1) - t(n)$ as follows:

$$\frac{dv}{dt} \approx \frac{v(n+1) - v(n)}{\Delta t} = g - k\left(\frac{v(n+1) + v(n)}{2}\right),$$

where the value identified by the integer n is the value of v at the beginning of the time interval, i.e. $v(n)$ at $t(n)$. The value identified by $n+1$ is the value of v at the end of the time interval, i.e. $v(n+1)$ at $t(n+1)$. This is an initial-value problem; hence, the value of v is known at $t(n)$. Knowing $v(n)$ at $t(n)$, and by specifying $t(n+1)$, the value $v(n+1)$ can be calculated by solving the finite-difference equation for $v(n+1)$ given above. The solution will depend on the size of Δt. This formula gives us $v(n+1)$. We next need to solve for $y(n+1)$ from the definition of v (viz., an approximate form of dy/dt). The average value of v over the interval of time, Δt, can be written in terms of y as follows:

$$\frac{dy}{dt} \approx \frac{y(n+1) - y(n)}{\Delta t} = \frac{v(n+1) + v(n)}{2}.$$

Rearranging this equation, we get a formula for $y(n+1)$ in terms of $v(n+1)$, $v(n)$, $y(n)$ and Δt. Given the initial condition, i.e. $y(n)$ and $v(n)$ at $t(n)$, we compute $v(n+1)$ with the first difference equation followed by computing $y(n+1)$ with the last equation. Note *that the new values become the initial condition for the next time step.* This procedure is done repeatedly until the overall time duration of interest is reached (in this case this is $t = 5$). The script file to implement this procedure is given next. The *m-file* is executed and the results plotted on the same graph with the exact solution. As shown, the comparison is remarkable. This is not always the case when approximate methods are used. The comparison is, of course, encouraging because we do not usually have exact results and need to resort to numerical approximations in our work.

The script file applied to examine the difference between the approximate method and the exact result is as follows:

```
%
% Approximate and exact solution comparison of
% free fall with linear friction for t = 0 to 5.
%
% Script by D. T. V. ...........September 2006.
%
% Step 1: Specified constants
%
k = 0.2;
g = 9.81;
%
% Step 2: Selection of time step for approximate solution
%           dt = t(n+1) - t(n)
dt = 0.01;
%
% Step 3: Initial condition
%
t(1) = 0.;
v(1) = 0.;
y(1) = 0.;
%
% Step 3: Sequential implementation of the approximate
% solution method by repeating the procedure 500 times
    %(using a for loop).
%
for n = 1:1:500
    t(n+1) = t(n) + dt;
    v(n+1) = (v(n) + dt * (g-0.5*k*v(n)) )/(1.+dt*0.5*k);
    y(n+1) = y(n) + dt * 0.5 * (v(n+1) + v(n));
end
%
% Step 4: Exact solution over the same interval of time:
%
    ye = (g/k) * t - (g/(k^2)) * (1.-exp(-k*t));
%
% Step 5: Graphical comparison:
%
plot(t,y,'o',t,ye);title('Comparison of numerics w/
    exact results'), ... xlabel(' Time, t'),ylabel
    (' Distance of free fall from rest, y')
%
% Step 6: Comparison of distance at t=5
%
disp(' ');
```

```
fprintf(' y(t) = %f, ye(t) = %f at t = %f \n',y(501),
  ye(501),t(501))
%
% Step 7: End of script by DTV.
%
```

In summary, the examination of the free-fall problem illustrated the application and the checking of formulae reported in the literature. In addition, an approximate method was applied to solve the same problem to illustrate the application of a method that would be necessary if exact formulae were not found. Finally, the approximate method that was applied can certainly be improved by utilizing the ordinary differential equation solvers available in MATLAB. Examples of this type of procedure are given in the last chapter. Other examples are in the help manuals in MATLAB that are found via the question mark (?) in the toolbar on the desktop window system (as already mentioned).

14.4 Projectile with friction

Let us examine the projectile problem again. In this case let us examine the effect of air resistance on the flight of a projectile like a golf ball. For a unit mass, the formulae that describe the trajectory of a projectile in a gravitational field, g, with air resistance in the opposite direction of motion (that is proportional to the speed of the projectile in the direction of motion squared) are as follows:

$$\begin{aligned} u &= dx/dt, \\ v &= dy/dt, \\ du/dt &= -ku * \sqrt{u^2 + v^2}, \\ dv/dt &= -kv * \sqrt{u^2 + v^2} - g. \end{aligned}$$

The location of the projectile (e.g. a golf ball) is initially at $x = 0$, $y = 0$. It is launched at a speed V_s in the direction of angle θ as measured from the horizontal. The coordinate x is in the horizontal direction parallel to the ground. The coordinate y is in the direction perpendicular to x pointing towards the sky. Hence, the gravitational acceleration is in the negative y direction. For a given launch speed and direction, we wish to estimate the range (or the distance) the ball travels in the x direction when it first hits the ground. To do this we shall approximate the four equations in a similar way that we treated the free-falling object problem in the last section.

The solution method is given in detail in the script provided at the end of this paragraph. It was saved with the name `golf.m`.

```
%
% "The Golf ball problem"
% Numerical computation of the trajectory of a
% projectile launched at an angle theta with
% a specified launch speed. They are:
%  theta = launch angle in degrees.
%  Vs = launch speed.
%
%  Script by D. T. V. ..... September 2006
%
% Equations of motion:
%  u = dx/dt.  v = dy/dt. g is in the opposite
%  direction of y. x = y = 0 is the location of
%  the tee. k is the coefficient of air drag. It
%  is assumed to be constant. Friction is assumed
%  to be proportional to the speed of the ball
%  squared and it acts in the opposite direction
%  of the direction of motion of the ball. The
%  components of acceleration are thus:
%  du/dt = - [k (u^2 + v^2) * u/sqrt(u^2+v^2)].
%  dv/dt = - [k (u^2 + v^2) * v/sqrt(u^2+v^2)] - g.
%
% INPUT DATA
% Specified constants:
k = 0.02;
g = 9.81;
dt = 0.01;
%
% Input the initial condition:
%
theta = input(' Initial angle of launch: ')
the = theta * pi/180.;
Vs = input('  Initial speed of launch:  ')
u(1) = Vs * cos(the);
v(1) = Vs * sin(the);
% Launch pad location:
x(1) = 0.;
y(1) = 0.;
%
% Compute approximate solution of the trajectory of
    flight.
% Repeat up to 6000 time, i.e. until ball hits the
    ground.
%
for n=1:1:6000;
     u(n+1) = u(n) - dt * (k * sqrt(u(n)^2+v(n)^2) *
                           u(n));
```

```
     v(n+1) = v(n) - dt * (k * sqrt(u(n)^2+v(n)^2) *
                           v(n) + g);
     x(n+1) = x(n) + u(n) * dt;
     y(n+1) = y(n) + v(n) * dt;
% Determination of when the object hits ground:
  if y(n+1) < 0
      slope = (y(n+1) - y(n))/(x(n+1) - x(n));
      b = y(n) - slope * x(n);
      xhit = - b/slope;
    plot(x,y)
    fprintf(' The length of the shot =  %5.2f \n', xhit)
  end
% Once object hits terminate the computations with
    %a break:
    if y(n+1) < 0; break; end
end
%
% Graphical presentation of the results:
%
if y(n+1) > 0
    plot(x,y)
end
% ------ End of golf-ball script by DTV
```

What is the optimum launch angle for $k = 0.02$? The answer to this question is described next for the launch speed of 100. Note that the optimum angle is the angle which gives the longest distance of travel (or greatest range). To compute the optimum angle the following script was executed after the lines with the input statements in the golf.m scripted were commented out by typing % at the beginning of the two lines with the `input` commands. The two lines in question were commented out as follows:

```
%
% In golf.m the following alterations were made
% prior to running this script!!!!!!
%
% % theta = input(' Initial angle of launch: ')
% % Vs = input('  Initial speed of launch:  ')
%
%  This script then finds the optimum angle for k = 0.2
%  to compare with the zero friction case, which we know
%  has the optimum launch angle of 45 degrees.
%
% Script by D. T. V. .....September 2006
%
% consider launch angles from 1 to 45 degrees
```

```
%
th = 1:1:45;
vs = 100;          % Specified launch speed.
%
% Execute the modified golf.m file 45 times and save
% results for each execution of golf.m
%
for i=1:45
    theta = th(i)
    golf % Execution of modified golf.m script.
    xh(i) = xhit
    thxh(i) = theta
end
% Find the maximum distance and the corresponding index
[xmh,n] = max(xh)
% Determine the angle that the maximum distance occurred.
opt_angle = thxh(n)
$ Display the results
disp(' optimum angle ')
disp( opt_angle )
% REMARK: For this case the result is 30 degrees.
% End of script
```

The optimum angle of launch for the case of nonlinear friction was computed and found to be equal to 30 degrees. Without friction it is 45 degrees. Hence, it is not surprising that golfers launch their best drives at angles significantly less than 45 degrees.

Summary

In this chapter we examined three problems using some of the utilities in MATLAB. We determined the following:

- We determined the shape of the deflection of a cantilever beam based on a published formula in the engineering literature. Thus, we could use the capability of MATLAB to do arithmetic with polynomials.
- We examined the effect of friction on the free-fall problem. In addition to computing the distance and speed of free fall based on published formulae, we checked the formulae with the symbolics tools and we also solved the problem by an approximate method to illustrate the range of possibilities at your disposal to solve technical problems.

➤ We examined the projectile problem subjected to air resistance. We did this by applying the same type of approximate method as we applied in the free-fall problem. We found that the optimum angle of launch with friction taken into account is less than 45 degrees (the frictionless value).

EXERCISES

14.1 Reproduce the script for the exact solutions for the free-fall problem (called FFall.m) and execute it for a range of friction coefficients, e.g. $k = 0.1$ and 0.3.

14.2 Use the symbolics tools in the same way they were used to check the formulae for the linear-friction case of free fall, to check the other two cases.

14.3 Reproduce the golf.m (projectile script) and look at the effect of varying the friction coefficient k. Look at $k = 0.01$ and 0.03. What influence does friction have on the optimum angle for a launch speed of 100?

15

Simulation

The objective of this chapter is to introduce you to:

- **Simulation of "real-life" events.**

Simulation is an area of application where computers have come into their own. A simulation is a *computer experiment* which mirrors some aspect of the real world that appears to be based on random processes, or is too complicated to understand properly. (Whether events can be really random is actually a philosophical or theological question.) Some examples are: radioactive decay, rolling dice, bacteria division and traffic flow. The essence of a simulation program is that the programmer is unable to predict beforehand exactly what the outcome of the program will be, which is true to the event being simulated. For example, when you spin a coin, you do not know for sure what the result will be.

15.1 Random number generation

Random events are easily simulated in MATLAB with the function `rand`, which we have briefly encountered already. By default, `rand` returns a *uniformly distributed pseudo-random* number in the range $0 \leq$ `rand` < 1. (A computer cannot generate truly random numbers, but they can be practically unpredictable.) `rand` can also generate row or column vectors, e.g. `rand(1,5)` returns a row vector of five random numbers (1 row, 5 column) as shown:

```
0.9501    0.2311    0.6068    0.4860    0.8913
```

If you generate more random numbers during the same MATLAB session, you will get a different sequence each time, as you would expect. However, each time you start a MATLAB session, the random number sequence begins at the same place (0.9501), and continues in the same way. This is not true to life,

as every gambler knows. To produce a different sequence each time you start a session, `rand` can be initially *seeded* in a different way each time you start.

15.1.1 Seeding `rand`

The random number generator `rand` can be seeded with the statement

```
rand('state', n)
```

where `n` is any integer. (By default, `n` is set to 0 when a MATLAB session starts.) This is useful if you want to generate the *same* random sequence every time a script runs, e.g. in order to debug it properly. Note that this statement does not generate any random numbers; it only initializes the generator.

However, you can also arrange for `n` to be different each time you start MATLAB by using the system time. The function `clock` returns the date and time in a six-element vector with seconds to two decimal places, so the expression `sum(100*clock)` never has the same value (well, hardly ever). You can use it to seed `rand` as follows:

```
≫ rand('state', sum(100*clock))
≫ rand(1,7)
ans =
  0.3637  0.2736  0.9910  0.3550  0.8501  0.0911  0.4493
≫ rand('state', sum(100*clock))
≫ rand(1,7)
ans =
  0.9309  0.2064  0.7707  0.7644  0.2286  0.7722  0.5315
```

Theoretically `rand` can generate over 2^{1492} numbers before repeating itself.

15.2 Spinning coins

When a fair (unbiased) coin is spun, the probability of getting heads or tails is 0.5 (50%). Since a value returned by `rand` is equally likely to anywhere in the interval [0, 1) we can represent heads, say, with a value less than 0.5, and tails otherwise.

Suppose an experiment calls for a coin to be spun 50 times, and the results recorded. In real life you may need to repeat such an experiment a number

of times; this is where computer simulation is handy. The following script simulates spinning a coin 50 times:

```
for i = 1:50
  r = rand;
  if r < 0.5
    fprintf( 'H' )
  else
    fprintf( 'T' )
  end
end
fprintf( '\n' )          % newline
```

Here is the output from two sample runs:

```
THHTTHHHHTTTTTHTHTTTHHTHTTTHHTTTTHTTHHTHHHHHHTTHTT
THTHHHTHTHHTTHTHTTTHHTTTTTTTHHHTTTHTHHTHHHHTTHTHTT
```

Note that it should be impossible in principle to tell from the output alone whether the experiment was simulated or real (if the random number generator is sufficiently random).

Can you see why it would be wrong to code the `if` part of the coin simulation like this:

```
if rand < 0.5 fprintf( 'H' ), end
if rand >= 0.5 fprintf( 'T' ), end
```

The basic principle is that `rand` should be called only *once* for each 'event' being simulated. Here the single event is spinning a coin, but `rand` is called twice. Also, since two different random numbers are generated, it is quite possible that *both* logical expressions will be true, in which case 'H' and 'T' will both be displayed for the same coin!

15.3 Rolling dice

When a fair dice is rolled, the number uppermost is equally likely to be any integer from 1 to 6. We saw in **Counting random numbers** (Chapter 5) how to

use `rand` to simulate this. The following statement generates a vector with 10 random integers in the range 1–6:

```
d = floor( 6 * rand(1,10) + 1 )
```

Here are the results of two such simulations:

```
2     1     5     5     6     3     4     5     1     1
4     5     1     3     1     3     5     4     6     6
```

We can do statistics on our simulated experiment, just as if it were a real one. For example, we could estimate the mean of the number obtained when the dice is rolled 100 times, and the probability of getting a six, say.

15.4 Bacteria division

If a fair coin is spun, or a fair dice is rolled, the different events (e.g. getting 'heads', or a 6) happen with equal likelihood. Suppose, however, that a certain type of bacteria divides (into two) in a given time interval with a probability of 0.75 (75%), and that if it does not divide, it dies. Since a value generated by `rand` is equally likely to be anywhere between 0 and 1, the chances of it being *less than* 0.75 are precisely 75%. We can therefore simulate this situation as follows:

```
r = rand;
if r < 0.75
  disp( 'I am now we' )
else
  disp( 'I am no more' )
end
```

Again, the basic principle is that one random number should be generated for each event being simulated. The single event here is the bacterium's life history over the time interval.

15.5 A random walk

A seriously short-sighted sailor has lost his contact lenses returning from a symphony concert, and has to negotiate a jetty to get to his ship. The jetty is 50 paces long and 20 wide. He is in the middle of the jetty at the quay-end,

pointing toward the ship. Suppose at every step he has a 60 percent chance of stumbling blindly toward the ship, but a 20 percent chance of lurching to the left or right (he manages to be always facing the ship). If he reaches the ship-end of the jetty, he is hauled aboard by waiting mates.

The problem is to simulate his progress along the jetty, and to estimate his chances of getting to the ship without falling into the sea. To do this correctly, we must simulate one *random walk* along the jetty, find out whether or not he reaches the ship, and then repeat this simulation 1000 times, say (if we have a fast enough computer!). The proportion of simulations that end with the sailor safely in the ship will be an estimate of his chances of making it to the ship. For a given walk we assume that if he has not either reached the ship or fallen into the sea after, say, 10 000 steps, he dies of thirst on the jetty.

To represent the jetty, we set up coordinates so that the x-axis runs along the middle of the jetty with the origin at the quay-end. x and y are measured in steps. The sailor starts his walk at the origin each time. The structure plan and script are as follows:

1. Initialize variables, including number of walks n
2. Repeat n simulated walks down the jetty:
    Start at the quay-end of the jetty
    While still on the jetty and still alive repeat:
        Get a random number R for the next step
        If $R < 0.6$ then
            Move forward (to the ship)
        Else if $R < 0.8$ then
            Move port (left)
        Else
            Move starboard
    If he got to the ship then
        Count that walk as a success
3. Compute and print estimated probability of reaching the ship
4. Stop.

```
% random walk
n = input( 'Number of walks: ' );
nsafe = 0;                    % number of times he makes it

for i = 1:n
  steps = 0;                  % each new walk ...
  x = 0;                      % ... starts at the origin
  y = 0;
```

```
  while x <= 50 & abs(y) <= 10 & steps < 1000
    steps = steps + 1;        % that's another step
    r = rand;                 % random number for that step
    if r < 0.6                % which way did he go?
      x = x + 1;              % maybe forward ...
    elseif r < 0.8
      y = y + 1;              % ... or to port ...
    else
      y = y - 1;              % ... or to starboard
    end;
  end;

  if x > 50
    nsafe = nsafe + 1;        % he actually made it this
                                time!
  end;

end;

prob = 100 * nsafe / n;
disp( prob );
```

A sample run of 100 walks took about 1.7 seconds on Hahn's Celeron, and gave an 93% probability of reaching the ship.

You can speed up the script by about 20% if you generate a vector of 1000 random numbers, say, at the start of each walk (with `r = rand(1,1000);`) and then reference elements of the vector in the `while` loop, e.g.

```
if r(steps) < 0.6 ...
```

15.6 Traffic flow

A major application of simulation is in modeling the traffic flow in large cities, in order to test different traffic light patterns before inflicting them on the real traffic. In this example we look at a very small part of the problem: how to simulate the flow of a single line of traffic through one set of traffic lights. We make the following assumptions (you can make additional or different ones if you like):

1. Traffic travels straight, without turning.
2. The probability of a car arriving at the lights in a particular second is independent of what happened during the previous second. This is called a *Poisson process*. This probability (call it p) may be estimated by watching cars at

the intersection and monitoring their arrival pattern. In this simulation we take $p = 0.3$.

3. When the lights are green, assume the cars move through at a steady rate of, say, eight every ten seconds.
4. In the simulation, we will take the basic time period to be ten seconds, so we want a display showing the length of the queue of traffic (if any) at the lights every ten seconds.
5. We will set the lights red or green for variable multiples of ten seconds.

The situation is modeled with a script file `traffic.m` which calls three function files: `go.m`, `stop.m`, and `prq.m`. Because the function files need access to a number of base workspace variables created by `traffic.m`, these variables are declared `global` in `traffic.m`, and in all three function files.

In this example the lights are red for 40 seconds (`red = 4`) and green for 20 seconds (`green = 2`). The simulation runs for 240 seconds (`n = 24`).

The script, `traffic.m`, is as follows:

```
clc
clear                  % clear out any previous garbage!
global CARS GTIMER GREEN LIGHTS RED RTIMER T

CARS = 0;              % number of cars in queue
GTIMER = 0;            % timer for green lights
GREEN = 2;             % period lights are green
LIGHTS = 'R';          % color of lights
n = 48;                % number of 10-sec periods
p = 0.3;               % probability of a car arriving
RED = 4;               % period lights are red
RTIMER = 0;            % timer for red lights

for T = 1:n            % for each 10-sec period

  r = rand(1,10);      % 10 seconds means 10 random
                         numbers
  CARS = CARS + sum(r < p);  % cars arriving in 10
                               seconds

  if LIGHTS == 'G'
    go                 % handles green lights
  else
    stop               % handles red lights
  end;

end;
```

The function files `go.m`, `stop.m` and `prq.m` (all separate M-files) are as follows:

```
% ------------------------------------------------------
function go
global CARS GTIMER GREEN LIGHTS
GTIMER = GTIMER + 1;        % advance green timer
CARS = CARS - 8;            % let 8 cars through

if CARS < 0                 % ... there may have been < 8
  CARS = 0;
end;

prq;                        % display queue of cars

if GTIMER == GREEN          % check if lights need to
                              change
  LIGHTS = 'R';
  GTIMER = 0;
end;


% ------------------------------------------------------
function stop
global LIGHTS RED RTIMER
RTIMER = RTIMER + 1;       % advance red timer
prq;                       % display queue of cars

if RTIMER == RED           % check if lights must be
                             changed
  LIGHTS = 'G';
  RTIMER = 0;
end;


% ------------------------------------------------------
function prq
global CARS LIGHTS T
fprintf( '%3.0f ', T );    % display period number

if LIGHTS == 'R'           % display color of lights
  fprintf( 'R    ' );
else
  fprintf( 'G    ' );
end;
for i = 1:CARS             % display * for each car
   fprintf( '*' );
```

```
end;

fprintf( '\n' )             % new line
```

Typical output looks like this:

```
 1 R    ****
 2 R    ********
 3 R    ***********
 4 R    **************
 5 G    **********
 6 G    *****
 7 R    ********
 8 R    *************
 9 R    ****************
10 R    ******************
11 G    **************
12 G    **********
13 R    **************
14 R    ****************
15 R    ********************
16 R    ************************
17 G    **********************
18 G    ****************
19 R    ******************
20 R    **********************
21 R    ************************
22 R    ******************************
23 G    ************************
24 G    **********************
```

From this particular run it seems that a traffic jam is building up, although more and longer runs are needed to see if this is really so. In that case, one can experiment with different periods for red and green lights in order to get an acceptable traffic pattern before setting the real lights to that cycle. Of course, we can get closer to reality by considering two-way traffic, and allowing cars to turn in both directions, and occasionally to break down, but this program gives the basic ideas.

15.7 Normal (Gaussian) random numbers

The function `randn` generates *Gaussian* or *normal* random numbers (as opposed to *uniform*) with a mean (μ) of 0 and a variance (σ^2) of 1.

- Generate 100 normal random numbers `r` with `randn(1,100)` and draw their histogram. Use the functions `mean(r)` and `std(r)` to find their mean and standard deviation (σ).
- Repeat with 1000 random numbers. The mean and standard deviation should be closer to 0 and 1 this time.

The functions `rand` and `randn` have separate generators, each with its own seed.

Summary

- A simulation is a computer program written to mimic a "real-life" situation which is apparently based on chance.
- The pseudo-random number generator `rand` returns uniformly distributed random numbers in the range [0, 1), and is the basis of the simulations discussed in this chapter.
- `randn` generates normally distributed (Gaussian) random numbers.
- `rand('state', n)` enables the user to seed `rand` with any integer `n`. A seed may be obtained from `clock`, which returns the system clock time.
  `randn` may be seeded (independently) in a similar way.
- Each independent event being simulated requires one and only one random number.

EXERCISES

15.1 Write some statements to simulate spinning a coin 50 times using 0-1 vectors instead of a `for` loop. **Hints:** Generate a vector of 50 random numbers, set up 0-1 vectors to represent the heads and tails, and use `double` and `char` to display them as a string of `H`s and `T`s.

15.2 In a game of Bingo the numbers 1 to 99 are drawn at random from a bag. Write a script to simulate the draw of the numbers (each number can be drawn only once), printing them ten to a line.

15.3 Generate some strings of 80 random alphabetic letters (lowercase only). For fun, see how many real words, if any, you can find in the strings.

15.4 A random number generator can be used to estimate π as follows (such a method is called a *Monte Carlo* method). Write a script which generates

random points in a square with sides of length 2, say, and which counts what proportion of these points falls inside the circle of unit radius that fits exactly into the square. This proportion will be the ratio of the area of the circle to that of the square. Hence estimate π. (This is not a very efficient method, as you will see from the number of points required to get even a rough approximation.)

15.5 Write a script to simulate the progress of the short-sighted student in Chapter 16 (**Markov Processes**). Start him at a given intersection, and generate a random number to decide whether he moves toward the internet cafe or home, according to the probabilities in the transition matrix. For each simulated walk, record whether he ends up at home or in the cafe. Repeat a large number of times. The proportion of walks that end up in either place should approach the limiting probabilities computed using the Markov model described in Chapter 16. **Hint:** If the random number is less than 2/3 he moves toward the cafe (unless he is already at home or in the cafe, in which case that random walk ends), otherwise he moves toward home.

15.6 The aim of this exercise is to simulate bacteria growth.

Suppose that a certain type of bacteria divides or dies according to the following assumptions:

(a) during a fixed time interval, called a *generation*, a single bacterium divides into two identical replicas with probability p;

(b) if it does not divide during that interval, it dies;

(c) the offspring (called daughters) will divide or die during the next generation, independently of the past history (there may well be no offspring, in which case the colony becomes extinct).

Start with a single individual and write a script which simulates a number of generations. Take $p = 0.75$. The number of generations which you can simulate will depend on your computer system. Carry out a large number (e.g. 100) of such simulations. The probability of ultimate extinction, $p(E)$, may be estimated as the proportion of simulations that end in extinction. You can also estimate the mean size of the nth generation from a large number of simulations. Compare your estimate with the theoretical mean of $(2p)^n$.

Statistical theory shows that the expected value of the extinction probability $p(E)$ is the smaller of 1, and $(1-p)/p$. So for $p = 0.75$, $p(E)$ is expected to be 1/3. But for $p \leq 0.5$, $p(E)$ is expected to be 1, which means that extinction is certain (a rather unexpected result). You can use your script to test this theory by running it for different values of p, and estimating $p(E)$ in each case.

15.7 Brian Hahn (the original author of this book) is indebted to a colleague, Gordon Kass, for suggesting this problem.

Dribblefire Jets Inc. make two types of airplane, the two-engined DFII, and the four-engined DFIV. The engines are terrible and fail with probability 0.5 on a standard flight (the engines fail independently of each other). The manufacturers claim that the planes can fly if at least half of their engines are working, i.e. the DFII will crash only if both its engines fail, while the DFIV will crash if all four, or if any three engines fail.

You have been commissioned by the Civil Aviation Board to ascertain which of the two models is less likely to crash. Since parachutes are expensive, the cheapest (and safest!) way to do this is to simulate a large number of flights of each model. For example, two calls of `Math.random` could represent one standard DFII flight: if both random numbers are less than 0.5, that flight crashes, otherwise it doesn't. Write a script which simulates a large number of flights of both models, and estimates the probability of a crash in each case. If you can run enough simulations, you may get a surprising result. (Incidentally, the probability of n engines failing on a given flight is given by the binomial distribution, but you do not need to use this fact in the simulation.)

15.8 Two players, *A* and *B*, play a game called *Eights*. They take it in turns to choose a number 1, 2 or 3, which may not be the same as the last number chosen (so if *A* starts with 2, *B* may only choose 1 or 3 at the next move). *A* starts, and may choose any of the three numbers for the first move. After each move, the number chosen is added to a common running total. If the total reaches 8 exactly, the player whose turn it was wins the game. If a player causes the total to go over 8, the other player wins. For example, suppose *A* starts with 1 (total 1), *B* chooses 2 (total 3), *A* chooses 1 (total 4) and *B* chooses 2 (total 6). *A* would like to play 2 now, to win, but he can't because *B* cunningly played it on the last move, so *A* chooses 1 (total 7). This is even smarter, because *B* is forced to play 2 or 3, making the total go over 8 and thereby losing.

Write a script to simulate each player's chances of winning, if they always play at random.

15.9 If r is a normal random number with mean 0 and variance 1 (as generated by `randn`), it can be transformed into a random number X with mean μ and standard deviation σ by the relation

$$X = \sigma r + \mu.$$

In an experiment a Geiger counter is used to count the radioactive emissions of cobalt 60 over a 10-second period. After a large number of such readings are taken, the count rate is estimated to be normally distributed with a mean of 460 and a standard deviation of 20.

1. Simulate such an experiment 200 times by generating 200 random numbers with this mean and standard deviation. Plot the histogram (use 10 bins).
2. Repeat a few times to note how the histogram changes each time.

15.10 Radioactive carbon 11 has a decay-rate k of 0.0338 per minute, i.e. a particular C^{11} atom has a 3.38% chance of decaying in any one minute.

Suppose we start with 100 such atoms. We would like to simulate their fate over a period of 100 minutes, say. We want to end up with a bar graph showing how many atoms remain undecayed after 1, 2, …, 100 minutes.

We need to simulate when each of the 100 atoms decays. This can be done, for each atom, by generating a random number r for each of the 100 minutes, until either $r > k$ (that atom decays), or the 100 minutes is up. If the atom decayed at time $t < 100$, increment the frequency distribution $f(t)$ by 1. $f(t)$ will be the number of atoms decaying at time t minutes.

Now convert the number $f(t)$ decaying each minute to the number $R(t)$ *remaining* each minute. If there are n atoms to start with, after one minute, the number $R(1)$ remaining will be $n - f(1)$, since $f(1)$ is the number decaying during the first minute. The number $R(2)$ remaining after two minutes will be $n - f(1) - f(2)$. In general, the number remaining after t minutes will be (in MATLAB notation)

```
R(t) = n - sum( f(1:t) )
```

Write a script to compute $R(t)$ and plot its bar graph. Superimpose on the bar graph the theoretical result, which is

$$R(t) = 100 \exp^{-kt}.$$

Typical results are shown in Figure 15.1.

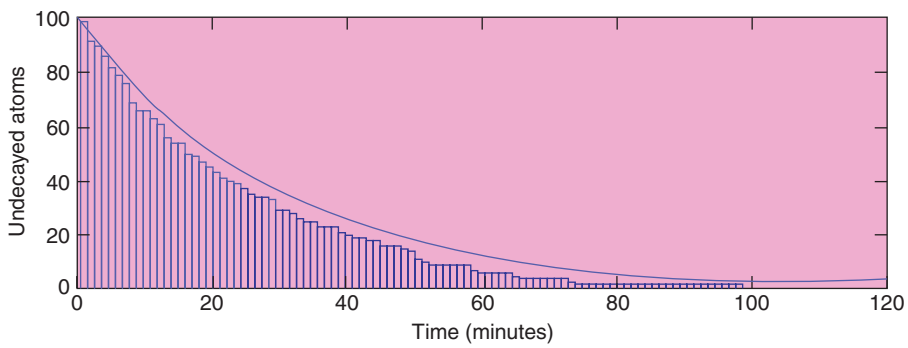


Figure 15.1 Radioactive decay of carbon 11: simulated and theoretical

16

*More matrices

The objectives of this chapter are to demonstrate the use of matrices in a number of application areas, including:

- **Population dynamics.**
- **Markov processes.**
- **Linear equations.**

And to introduce:

- **MATLAB's sparse matrix facilities.**

16.1 Leslie matrices: population growth

Our first application of matrices is in population dynamics.

Suppose we want to model the growth of a population of rabbits, in the sense that given their number at some moment, we would like to estimate the size of the population in a few years' time. One approach is to divide the rabbit population up into a number of age classes, where the members of each age class are one time unit older than the members of the previous class, the time unit being whatever is convenient for the population being studied (days, months, etc.).

If X_i is the size of the ith age class, we define a *survival factor* P_i as the proportion of the ith class that survive to the $(i+1)$th age class, i.e. the proportion that 'graduate'. F_i is defined as the *mean fertility* of the ith class. This is the mean number of newborn individuals expected to be produced during one time interval by each member of the ith class at the beginning of the interval (only females count in biological modeling, since there are always enough males to go round!).

Suppose for our modified rabbit model we have three age classes, with X_1, X_2 and X_3 members, respectively. We will call them young, middle-aged and old-aged for convenience. We will take our time unit as one month, so X_1 is the number that were born during the current month, and which will be considered as youngsters at the end of the month. X_2 is the number of middle-aged rabbits at the end of the month, and X_3 the number of oldsters. Suppose the youngsters cannot reproduce, so that $F_1 = 0$. Suppose the fertility rate for middle-aged rabbits is 9, so $F_2 = 9$, while for oldsters $F_3 = 12$. The probability of survival from youth to middle-age is one third, so $P_1 = 1/3$, while no less than half the middle-aged rabbits live to become oldsters, so $P_2 = 0.5$ (we are assuming for the sake of illustration that all old-aged rabbits die at the end of the month—this can be corrected easily). With this information we can quite easily compute the changing population structure month by month, as long as we have the population breakdown to start with.

If we now denote the current month by t, and next month by $(t+1)$, we can refer to this month's youngsters as $X_1(t)$, and to next month's as $X_1(t+1)$, with similar notation for the other two age classes. We can then write a scheme for updating the population from month t to month $(t+1)$ as follows:

$$\begin{aligned} X_1(t+1) &= F_2 X_2(t) + F_3 X_3(t), \\ X_2(t+1) &= P_1 X_1(t), \\ X_3(t+1) &= P_2 X_2(t). \end{aligned}$$

We now define a population vector $\mathbf{X}(t)$, with three components, $X_1(t)$, $X_2(t)$, and $X_3(t)$, representing the three age classes of the rabbit population in month t. The above three equations can then be rewritten as

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}_{(t+1)} = \begin{bmatrix} 0 & F_2 & F_3 \\ P_1 & 0 & 0 \\ 0 & P_2 & 0 \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}_t$$

where the subscript at the bottom of the vectors indicates the month. We can write this even more concisely as the matrix equation

$$\mathbf{X}(t+1) = \mathbf{L}\,\mathbf{X}(t), \tag{16.1}$$

where $\mathbf{L}$ is the matrix

$$\begin{bmatrix} 0 & 9 & 12 \\ 1/3 & 0 & 0 \\ 0 & 1/2 & 0 \end{bmatrix}$$

in this particular case. **L** is called a *Leslie matrix*. A population model can always be written in the form of Equation (16.1) if the concepts of age classes, fertility, and survival factors, as outlined above, are used.

Now that we have established a matrix representation for our model, we can easily write a script using matrix multiplication and repeated application of Equation (16.1):

$$\mathbf{X}(t+2) = \mathbf{L}\,\mathbf{X}(t+1),$$
$$\mathbf{X}(t+3) = \mathbf{L}\,\mathbf{X}(t+2), \text{ etc.}$$

We will assume to start with that we have one old (female) rabbit, and no others, so $X_1 = X_2 = 0$, and $X_3 = 1$. Here is the script:

```
% Leslie matrix population model
n = 3;
L = zeros(n);   % all elements set to zero
L(1,2) = 9;
L(1,3) = 12;
L(2,1) = 1/3;
L(3,2) = 0.5;
x = [0 0 1]';   % remember x must be a column vector!

for t = 1:24
  x = L * x;
  disp( [t x' sum(x)] )   % x' is a row
end
```

The output, over a period of 24 months (after some editing), is:

```
Month       Young     Middle        Old       Total

    1          12          0          0          12
    2           0          4          0           4
    3          36          0          2          38
    4          24         12          0          36
    5         108          8          6         122
  ...
   22    11184720    1864164     466020    13514904
   23    22369716    3728240     932082    27030038
   24    44739144    7456572    1864120    54059836
```

It so happens that there are no 'fractional' rabbits in this example. If there are any, they should be kept, and not rounded (and certainly not truncated). They occur because the fertility rates and survival probabilities are averages.

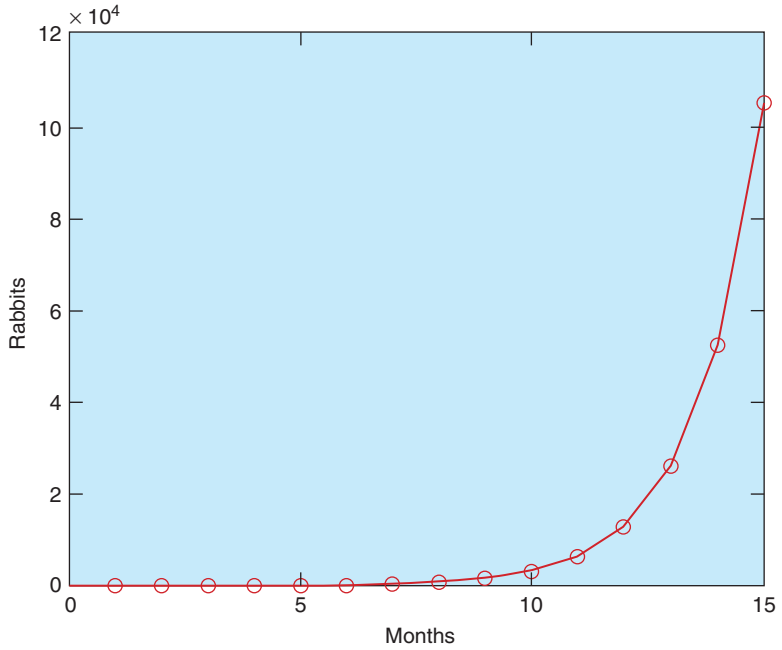


Figure 16.1 Total rabbit population over 15 months

If you look carefully at the output you may spot that after some months the total population doubles every month. This factor is called the *growth factor*, and is a property of the particular Leslie matrix being used (for those who know about such things, it's the *dominant eigenvalue* of the matrix). The growth factor is 2 in this example, but if the values in the Leslie matrix are changed, the long-term growth factor changes too (try it and see).

Figure 16.1 shows how the total rabbit population grows over the first 15 months. To draw this graph yourself, insert the line

```
p(t) = sum(x);
```

in the `for` loop after the statement `x = L * x;`, and run the program again. The vector `p` will contain the total population at the end of each month. Then enter the commands

```
plot(1:15, p(1:15)), xlabel('months'), ylabel('rabbits')
hold, plot(1:15, p(1:15),'o')
```

The graph demonstrates *exponential* growth. If you plot the population over the full 24-month period, you will see that the graph gets much steeper. This is a feature of exponential growth.

You probably didn't spot that the numbers in the three age classes tend to a limiting ratio of 24:4:1. This can be demonstrated very clearly if you run the model with an initial population structure having this limiting ratio. The limiting ratio is called the *stable age distribution* of the population, and again it is a property of the Leslie matrix (in fact, it is the *eigenvector* belonging to the dominant eigenvalue of the matrix). Different population matrices lead to different stable age distributions.

The interesting point about this is that a given Leslie matrix always eventually gets a population into the *same* stable age distribution, which increases eventually by the *same* growth factor each month, *no matter what the initial population breakdown is*. For example, if you run the above model with any other initial population, it will always eventually get into a stable age distribution of 24:4:1 with a growth factor of 2 (try it and see).

See `help eig` if you're interested in using MATLAB to compute eigenvalues and eigenvectors.

16.2 Markov processes

Often a process that we wish to model may be represented by a number of possible *discrete* (i.e. discontinuous) states that describe the outcome of the process. For example, if we are spinning a coin, then the outcome is adequately represented by the two states 'heads' and 'tails' (and nothing in between). If the process is random, as it is with spinning coins, there is a certain probability of being in any of the states at a given moment, and also a probability of changing from one state to another. If the probability of moving from one state to another depends on the present state only, and not on any previous state, the process is called a *Markov chain*. The progress of the myopic sailor in Chapter 15 is an example of such a process. Markov chains are used widely in such diverse fields as biology and business decision-making, to name just two areas.

16.2.1 A random walk

This example is a variation on the random walk simulation of Chapter 15. A street has six intersections. A short-sighted student wanders down the street. His home is at intersection 1, and his favorite internet cafe at intersection 6. At each intersection other than his home or the cafe he moves in the direction of the cafe with probability 2/3, and in the direction of his home with probability 1/3. In other words, he is twice as likely to move towards the cafe as towards his home. He never wanders down a side street. If he reaches his home or the

cafe, he disappears into them, never to reappear (when he disappears we say in Markov jargon that he has been *absorbed*).

We would like to know: what are the chances of him ending up at home or in the cafe, if he starts at a given corner (other than home or the cafe, obviously)? He can clearly be in one of six states, with respect to his random walk, which can be labeled by the intersection number, where state 1 means *Home* and state 6 means *Cafe*. We can represent the probabilities of being in these states by a six-component *state vector* $\mathbf{X}(t)$, where $X_i(t)$ is the probability of him being at intersection i at moment t. The components of $\mathbf{X}(t)$ must sum to 1, since he has to be in one of these states.

We can express this Markov process with the following *transition probability matrix*, **P**, where the rows represent the next state (i.e. corner), and the columns represent the present state:

	Home	*2*	*3*	*4*	*5*	*Cafe*
Home	1	1/3	0	0	0	0
2	0	0	1/3	0	0	0
3	0	2/3	0	1/3	0	0
4	0	0	2/3	0	1/3	0
5	0	0	0	2/3	0	0
Cafe	0	0	0	0	2/3	1

The entries for *Home-Home* and *Cafe-Cafe* are both 1 because he stays there with certainty.

Using the probability matrix **P** we can work out his chances of being, say, at intersection 3 at moment $(t+1)$ as

$$X_3(t+1) = 2/3X_2(t) + 1/3X_4(t).$$

To get to 3, he must have been at either 2 or 4, and his chances of moving from there are 2/3 and 1/3, respectively.

Mathematically, this is identical to the Leslie matrix problem. We can therefore form the new state vector from the old one each time with a matrix equation:

$$\mathbf{X}(t+1) = \mathbf{P}\,\mathbf{X}(t).$$

If we suppose the student starts at intersection 2, the initial probabilities will be (0; 1; 0; 0; 0; 0). The Leslie matrix script may be adapted with very few changes to generate future states:

```
n = 6;
P = zeros(n);   % all elements set to zero

for i = 3:6
  P(i,i-1) = 2/3;
  P(i-2,i-1) = 1/3;
end

P(1,1) = 1;
P(6,6) = 1;
x = [0 1 0 0 0 0]'; % remember x must be a column vector!

for t = 1:50
  x = P * x;
  disp( [t x'] )
end
```

Edited output:

```
Time     Home        2        3        4        5     Cafe

   1   0.3333        0   0.6667        0        0        0
   2   0.3333   0.2222        0   0.4444        0        0
   3   0.4074        0   0.2963        0   0.2963        0
   4   0.4074   0.0988        0   0.2963        0   0.1975
   5   0.4403        0   0.1646        0   0.1975   0.1975
   6   0.4403   0.0549        0   0.1756        0   0.3292
   7   0.4586        0   0.0951        0   0.1171   0.3292
   8   0.4586   0.0317        0   0.1024        0   0.4073
 ...
  20   0.4829   0.0012        0   0.0040        0   0.5119
 ...
  40   0.4839   0.0000        0   0.0000        0   0.5161
 ...
  50   0.4839   0.0000        0   0.0000        0   0.5161
```

By running the program for long enough, we soon find the limiting probabilities: he ends up at home about 48 percent of the time, and at the cafe about 52 percent of the time. Perhaps this is a little surprising; from the transition probabilities, we might have expected him to get to the cafe rather more easily.

It just goes to show that you should never trust your intuition when it comes to statistics!

Note that the Markov chain approach is *not* a simulation: one gets the *theoretical* probabilities each time (this can all be done mathematically, without a computer). But it is interesting to confirm the limiting probabilities by *simulating* the student's progress, using a random number generator (see Exercise 15.5 at the end of Chapter 15).

16.3 Linear equations

A problem that often arises in scientific applications is the solution of a system of linear equations, e.g.

$$3x + 2y - z = 10 \tag{16.2}$$
$$-x + 3y + 2z = 5 \tag{16.3}$$
$$x - y - z = -1. \tag{16.4}$$

MATLAB was designed to solve a system like this directly and very easily, as we shall now see.

If we define the matrix **A** as

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix},$$

and the vectors **x** and **b** as

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix},$$

we can write the above system of three equations in matrix form as

$$\begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix},$$

or even more concisely as the single matrix equation

$$\mathbf{Ax} = \mathbf{b}. \tag{16.5}$$

The solution may then be written as

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}, \tag{16.6}$$

where $\mathbf{A}^{-1}$ is the matrix inverse of **A** (i.e. the matrix which when multiplied by **A** gives the identity matrix **I**).

16.3.1 MATLAB's solution

To see how MATLAB solves this system, first recall that the *left division* operator \ may be used on scalars, i.e. `a \ b` is the same as `b / a` if `a` and `b` are scalars. However, it can also be used on vectors and matrices, in order to solve linear equations. Enter the following statements on the command line to solve Equations (16.2)–(16.4):

```
A = [3 2 -1; -1 3 2; 1 -1 -1];
b = [10 5 -1]';
x = A \ b
```

This should result in

```
x =
   -2.0000
    5.0000
   -6.0000
```

In terms of our notation, this means that the solution is $x = -2, y = 5, z = -6$.

You can think of the matrix operation `A \ b` as 'b divided by A', or as 'the inverse of A multiplied by b', which is essentially what Equation (16.6) means. A colleague of mine reads this operation as 'A under b', which you may also find helpful.

You may be tempted to implement Equation (16.6) in MATLAB as follows:

```
x = inv(A) * b
```

since the function `inv` finds a matrix inverse directly. However, `A \ b` is actually more accurate and efficient, since it uses Gauss elimination, and therefore requires fewer *flops* (floating point operations), e.g. with a random 30×30 system `a \ b` needs 24051 flops, whereas `inv(a) * b` needs 59850—more than twice as many. (In case you're wondering where I got the number of flops from with such authority, earlier versions of MATLAB had a function `flops`

which counted the number of flops. In Version 7 this is no longer practical since the algorithms have been changed.)

16.3.2 The residual

Whenever we solve a system of linear equations numerically we need to have some idea of how accurate the solution is. The first thing to check is the *residual*, defined as

```
r = A*x - b
```

where `x` is the result of the operation `x = A \ b`. Theoretically the residual `r` should be zero, since the expression `A * x` is supposed to be equal to `b`, according to Equation (16.5), which is the equation we are trying to solve. In our example here the residual is (check it)

```
r =
   1.0e-015 *
         0
    0.8882
    0.6661
```

This seems pretty conclusive: all the elements of the residual are less than 10^{-15} in absolute value. Unfortunately, there may still be problems lurking beneath the surface, as we shall see shortly.

We will, however, first look at a situation where the residual turns out to be far from zero.

16.3.3 Overdetermined systems

When we have more equations than unknowns, the system is called *overdetermined*, e.g.

$$\begin{aligned} x - y &= 0 \\ y &= 2 \\ x &= 1. \end{aligned}$$

Surprisingly, perhaps, MATLAB gives a solution to such a system. If

```
A = [1 -1; 0 1; 1 0];
b = [0 2 1]';
```

the statement

```
x = A \ b
```

results in

```
x =
    1.3333
    1.6667
```

The residual `r = A*x - b` is now

```
r =
   -0.3333
   -0.3333
    0.3333
```

What happens in this case is that MATLAB produces the *least squares best fit*. This is the value of `x` which makes the *magnitude* of `r`, i.e.

$$\sqrt{\texttt{r(1)}^2 + \texttt{r(2)}^2 + \texttt{r(3)}^3},$$

as small as possible. You can compute this quantity (0.5774) with `sqrt(r' * r)` or `sqrt(sum(r .* r))`. There is a nice example of fitting a decaying exponential function to data with a least squares fit in **Using MATLAB: Mathematics: Matrices and Linear Algebra: Solving Linear Equations: Over-determined Systems**.

16.3.4 Underdetermined systems

If there are fewer equations than unknowns, the system is called *under-determined*. In this case there are an infinite number of solutions; MATLAB will find one which has zeros for some of the unknowns.

The equations in such a system are the constraints in a linear programming problem.

16.3.5 Ill conditioning

Sometimes the coefficients of a system of equations are the results of an experiment, and may be subject to error. We need in that case to know how

sensitive the solution is to the experimental errors. As an example, consider the system

$$\begin{aligned} 10x + 7y + 8z + 7w &= 32 \\ 7x + 5y + 6z + 5w &= 23 \\ 8x + 6y + 10z + 9w &= 33 \\ 7x + 5y + 9z + 10w &= 31 \end{aligned}$$

Use matrix left division to show that the solution is $x = y = z = w = 1$. The residual is exactly zero (check it), and all seems well.

However, if we change the right-hand side constants to 32.1, 22.9, 32.9 and 31.1, the 'solution' is now given by $x = 6, y = -7.2, z = 2.9, w = -0.1$. The residual is very small.

A system like this is called *ill-conditioned*, meaning that a small change in the coefficients leads to a large change in the solution. The MATLAB function `rcond` returns the *condition estimator*, which tests for ill conditioning. If `A` is the coefficient matrix, `rcond(A)` will be close to zero if `A` is ill-conditioned, but close to 1 if it is well-conditioned. In this example, the condition estimator is about 2×10^{-4}, which is pretty close to zero.

Some authors suggest the rule of thumb that a matrix is ill-conditioned if its determinant is small compared to the entries in the matrix. In this case the determinant of `A` is 1 (check with the function `det`) which is about an order of magnitude smaller than most of its entries.

16.3.6 Matrix division

Matrix left division, `A\B`, is defined whenever `B` has as many rows as `A`. This corresponds formally to `inv(A)*B` although the result is obtained without computing the inverse explicitly. In general,

```
x = A \ B
```

is a solution to the system of equations defined by $\mathbf{A}x = \mathbf{B}$.

If `A` is square, matrix left division is done using Gauss elimination.

If `A` is not square the over- or underdetermined equations are solved in the least squares sense. The result is a $m \times n$ matrix `X`, where m is the number of columns of `A` and n is the number of columns of `B`.

Matrix right division, `B/A`, is defined in terms of matrix left division such that `B/A` is the same as `(A'\B')'`. So with `a` and `b` defined as for Equations (16.2)–(16.4), this means that

```
x = (b'/a')'
```

gives the same solution, doesn't it? Try it, and make sure you can see why.

Sometimes the least squares solutions computed by `\` or `/` for over- or under-determined systems can cause surprises, since you can legally divide one *vector* by another. For example, if

```
a = [1 2];
b = [3 4];
```

the statement

```
a / b
```

results in

```
ans =
      0.4400
```

This is because `a/b` is the same as `(b'\a')'`, which is formally the solution of $\mathbf{b}'x' = \mathbf{a}'$. The result is a scalar, since `a'` and `b'` each have one column, and is the least squares solution of

$$\begin{pmatrix} 3 \\ 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

With this under your belt, can you explain why

```
a \ b
```

gives

```
ans =
           0         0
      1.5000    2.0000
```

(try writing the equations out in full)?

A complete discussion of the algorithms used in solving simultaneous linear equations may be found in the online documentation under **MATLAB: Reference: MATLAB Function Reference: Alphabetical List of Functions: Arithmetic Operators** `+ - * / \ ^ '`.

16.4 Sparse matrices

Matrices can sometimes get very large, with thousands of entries. Very large matrices can occupy huge amounts of memory, and processing them can take up a lot of computer time. For example, a system of n simultaneous linear equations requires n^2 matrix entries, and the computing time to solve them is proportional to n^3.

However, some matrices have relatively few *non-zero* entries. Such matrices are called *sparse* as opposed to *full*. The father of modern numerical linear algebra, J.H. Wilkinson, has remarked that a matrix is sparse if 'it has enough zeros that it pays to take advantage of them'. MATLAB has facilities for exploiting sparsity, which have the potential for saving huge amounts of memory and processing time. For example, the matrix representation of a certain type of partial differential equation (a 5-point Laplacian) on a square 64×64 grid is a 4096×4096 element matrix with 20 224 non-zero elements. The sparse form of this matrix in MATLAB occupies only 250 kB of memory, whereas the full version of the same matrix would occupy 128 MB, which is way beyond the limits of most desktop computers. The solution of the system $\mathbf{Ax} = \mathbf{b}$ using sparse techniques is about 4000 times faster than solving the full case, i.e. 10 seconds instead of 12 hours!

In this section (which you can safely skip, if you want to) we will look briefly at how to create sparse matrices in MATLAB. For a full description of sparse matrix techniques consult **Using MATLAB: Mathematics: Sparse Matrices**.

First an example, then an explanation. The transition probability matrix for the random walk problem in Section 16.2 is a good candidate for sparse representation. Only ten of its 36 elements are non-zero. Since the non-zeros appear only on the diagonal and the sub- and superdiagonals, a matrix representing more intersections would be even sparser. For example, a 100×100 representation of the same problem would have only 198 non-zero entries, i.e. 1.98%.

To represent a sparse matrix all that MATLAB needs to record are the non-zero entries with their row and column indices. This is done with the `sparse`

function. The transition matrix of Section 16.2 can be set up as a sparse matrix with the statements

```
n = 6;
P = sparse(1, 1, 1, n, n);
P = P + sparse(n, n, 1, n, n);
P = P + sparse(1:n-2, 2:n-1, 1/3, n, n);
P = P + sparse(3:n, 2:n-1, 2/3, n, n)
```

which result in (with `format rat`)

```
P =

   (1,1)      1
   (1,2)    1/3
   (3,2)    2/3
   (2,3)    1/3
   (4,3)    2/3
   (3,4)    1/3
   (5,4)    2/3
   (4,5)    1/3
   (6,5)    2/3
   (6,6)      1
```

Each line of the display of a sparse matrix gives a non-zero entry with its row and column, e.g. 2/3 in row 3 and column 2. To display a sparse matrix in full form, use the function

```
full(P)
```

which results in

```
ans =
     1   1/3     0     0     0     0
     0     0   1/3     0     0     0
     0   2/3     0   1/3     0     0
     0     0   2/3     0   1/3     0
     0     0     0   2/3     0     0
     0     0     0     0   2/3     1
```

(also with `format rat`).

The form of the `sparse` function used here is

```
sparse(rows, cols, entries, m, n)
```

This generates an `m` × `n` sparse matrix with non-zero `entries` having subscripts (`rows`, `cols`) (which may be vectors), e.g. the statement

```
sparse(1:n-2, 2:n-1, 1/3, n, n);
```

(with `n = 6`) creates a 6 × 6 sparse matrix with 4 non-zero elements, being 1/3's in rows 1–4 and columns 2–5 (most of the superdiagonal). Note that repeated use of `sparse` produces a number of 6 × 6 matrices, which must be added together to give the final form. Sparsity is therefore preserved by operations on sparse matrices. See `help` for more details of `sparse`.

It's quite easy to test the efficiency of sparse matrices. Construct a (full) identity matrix

```
a = eye(1000);
```

It takes about 27.5 seconds on Hahn's Celeron to compute `a^2`. However, `a` is an ideal candidate for being represented as a sparse matrix, since only 1000 of its one million elements are non-zero. It is represented in sparse form as

```
s = sparse(1:1000, 1:1000, 1, 1000, 1000);
```

It only takes about 0.05 seconds to compute `s^2`—more than 500 times faster!

The function `full(a)` returns the full form of the sparse matrix `a` (without changing the sparse representation of `a` itself). Conversely, `sparse(a)` returns the sparse form of the full matrix `a`.

The function `spy` provides a neat visualization of sparse matrices. Try it on `P`. Then enlarge `P` to about 50 × 50, and `spy` it.

Summary

- The matrix left division operator \ is used for solving systems of linear equations directly. Because the matrix division operators \ and \ can produce surprising results

with the least squares solution method, you should always compute the residual when solving a system of equations.

➤ If you work with large matrices with relatively few non-zero entries you should consider using MATLAB's sparse matrix facilities.

EXERCISES

16.1 Compute the limiting probabilities for the student in Section 16.2 when he starts at each of the remaining intersections in turn, and confirm that the closer he starts to the cafe, the more likely he is to end up there.

Compute `P^50` directly. Can you see the limiting probabilities in the first row?

16.2 Solve the equations

$$\begin{aligned} 2x - y + z &= 4 \\ x + y + z &= 3 \\ 3x - y - z &= 1 \end{aligned}$$

using the left division operator. Check your solution by computing the residual. Also compute the determinant (`det`) and the condition estimator (`rcond`). What do you conclude?

16.3 This problem, suggested by R.V. Andree, demonstrates ill conditioning (where small changes in the coefficients cause large changes in the solution). Use the left division operator to show that the solution of the system

$$\begin{aligned} x + 5.000y &= 17.0 \\ 1.5x + 7.501y &= 25.503 \end{aligned}$$

is $x = 2$, $y = 3$. Compute the residual.

Now change the term on the right-hand side of the second equation to 25.501, a change of about one part in 12 000, and find the new solution and the residual. The solution is completely different. Also try changing this term to 25.502, 25.504, etc. If the coefficients are subject to experimental errors, the solution is clearly meaningless. Use `rcond` to find the condition estimator and `det` to compute the determinant. Do these values confirm ill conditioning?

Another way to anticipate ill conditioning is to perform a *sensitivity analysis* on the coefficients: change them all in turn by the same small percentage, and observe what effect this has on the solution.

16.4 Use `sparse` to represent the Leslie matrix in Section 16.1. Test your representation by projecting the rabbit population over 24 months.

16.5 If you are familiar with *Gauss reduction* it is an excellent programming exercise to code a Gauss reduction directly with operations on the rows of the augmented coefficient matrix. See if you can write a function

x = mygauss(a, b)

to solve the general system $\mathbf{Ax} = \mathbf{b}$. Skillful use of the colon operator in the row operations can reduce the code to a few lines!

Test it on **A** and **b** with random entries, and on the systems in Section 16.3 and Exercise 16.4. Check your solutions with left division.

17

*Introduction to numerical methods

The objectives of this chapter are to introduce numerical methods for:

- **Solving equations.**
- **Evaluating definite integrals.**
- **Solving systems of ordinary differential equations.**
- **Solving a parabolic partial differential equation.**

A major scientific use of computers is in finding numerical solutions to mathematical problems which have no analytical solutions (i.e. solutions which may be written down in terms of polynomials and standard mathematical functions). In this chapter we look briefly at some areas where *numerical methods* have been highly developed, e.g. solving nonlinear equations, evaluating integrals, and solving differential equations.

17.1 Equations

In this section we consider how to solve equations in one unknown numerically. The usual way of expressing the problem is to say that we want to solve the equation $f(x) = 0$, i.e. we want to find its *root* (or roots). This process is also described as finding the *zeros* of $f(x)$. There is no general method for finding roots analytically for an arbitrary $f(x)$.

17.1.1 Newton's method

Newton's method is perhaps the easiest numerical method to implement for solving equations, and was introduced briefly in earlier chapters. It is an *iterative*

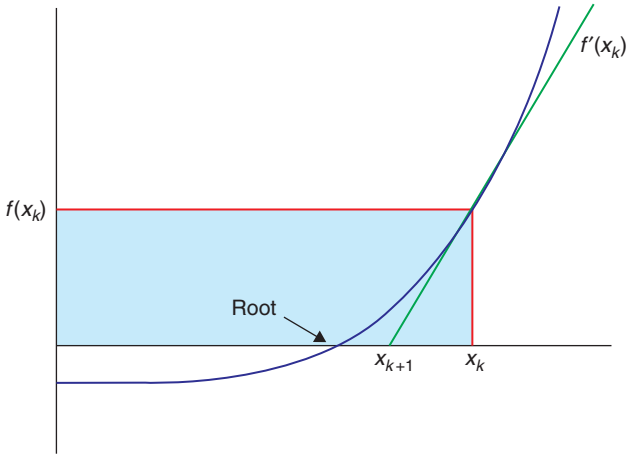


Figure 17.1 Newton's method

method, meaning that it repeatedly attempts to improve an estimate of the root. If x_k is an approximation to the root, we can relate it to the next approximation x_{k+1} using the right-angle triangle in Figure 17.1:

$$f'(x_k) = \frac{f(x_k) - 0}{x_k - x_{k+1}},$$

where $f'(x)$ is df/dx. Solving for x_{k+1} gives

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

A structure plan to implement Newton's method is:

1. Input starting value x_0 and required relative error e
2. While relative error $|(x_k - x_{k-1})/x_k| \geq e$ repeat up to $k = 20$, say:
   $x_{k+1} = x_k - f(x_k)/f'(x_k)$
   Print x_{k+1} and $f(x_{k+1})$
3. Stop.

It is necessary to limit step 2 since the process may not converge.

A script using Newton's method (without the subscript notation) to solve the equation $x^3 + x - 3 = 0$ is given in Chapter 10. If you run it you will see that the values of x converge rapidly to the root.

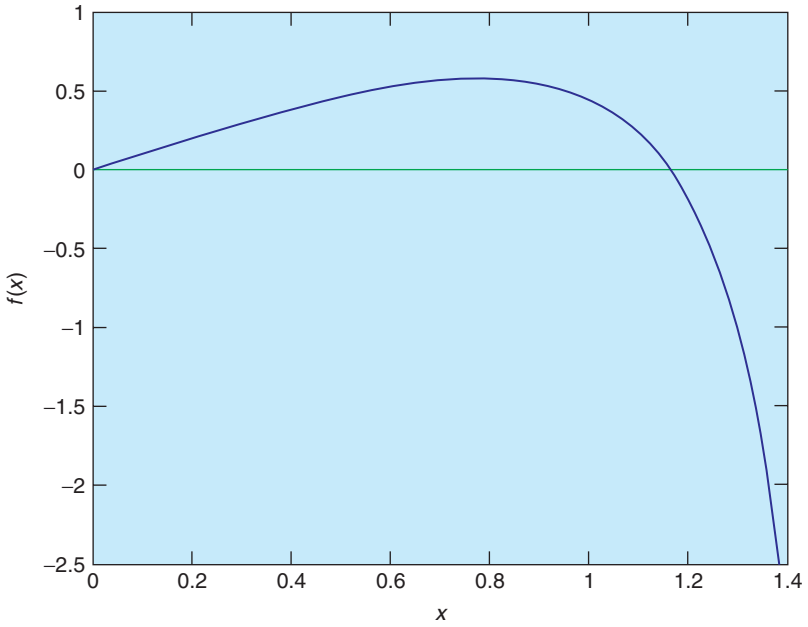


Figure 17.2 $f(x) = 2x - \tan(x)$

As an exercise, try running the script with different starting values of x_0 to see whether the algorithm always converges.

If you have a sense of history, use Newton's method to find a root of $x^3 - 2x - 5 = 0$. This is the example used when the algorithm was first presented to the French Academy.

Also try finding a *non-zero* root of $2x = tan(x)$, using Newton's method. You might have some trouble with this one. If you do, you will have discovered the one serious problem with Newton's method: it converges to a root only if the starting guess is 'close enough'. Since 'close enough' depends on the nature of $f(x)$ and on the root, one can obviously get into difficulties here. The only remedy is some intelligent trial-and-error work on the initial guess—this is made considerably easier by sketching $f(x)$ or plotting it with MATLAB (see Figure 17.2).

If Newton's method fails to find a root, the Bisection method, discussed below, can be used.

Complex roots

Newton's method can also find complex roots, but only if the starting guess is complex. Use the script in Chapter 10 to find a complex root of $x^2 + x + 1 = 0$.

Start with a complex value of `1 + i` say, for x. Using this starting value for x gives the following output (if you replace `disp( [x f(x)] )` in the script with `disp( x )`):

```
 0.0769 + 0.6154i
-0.5156 + 0.6320i
-0.4932 + 0.9090i
-0.4997 + 0.8670i
-0.5000 + 0.8660i
-0.5000 + 0.8660i
Zero found
```

Since complex roots occur in complex conjugate pairs, the other root is $-0.5 - 0.866i$.

17.1.2 The Bisection method

Consider again the problem of solving the equation $f(x) = 0$, where

$$f(x) = x^3 + x - 3.$$

We attempt to find by inspection, or trial and error, two values of x, call them x_L and x_R, such that $f(x_L)$ and $f(x_R)$ have different signs, i.e. $f(x_L)f(x_R) < 0$. If we can find two such values, the root must lie somewhere in the interval between them, since $f(x)$ changes sign on this interval (see Figure 17.3). In this example, $x_L = 1$ and $x_R = 2$ will do, since $f(1) = -1$ and $f(2) = 7$. In the Bisection method, we estimate the root by x_M, where x_M is the mid-point of the interval $[x_L, x_R]$, i.e.

$$x_M = (x_L + x_R)/2. \tag{17.1}$$

Then if $f(x_M)$ has the same sign as $f(x_L)$, as drawn in the figure, the root clearly lies between x_M and x_R. We must then redefine the left-hand end of the interval as having the value of x_M, i.e. we let the new value of x_L be x_M. Otherwise, if $f(x_M)$ and $f(x_L)$ have *different* signs, we let the new value of x_R be x_M, since the root must lie between x_L and x_M in that case. Having redefined x_L or x_R, as the case may be, we bisect the new interval again according to Equation (17.1) and repeat the process until the distance between x_L and x_R is as small as we please.

The neat thing about this method is that, *before* starting, we can calculate how many bisections are needed to obtain a certain accuracy, given initial values of x_L and x_R. Suppose we start with $x_L = a$, and $x_R = b$. After the first bisection the worst possible error (E_1) in x_M is $E_1 = |a - b|/2$, since we are estimating the

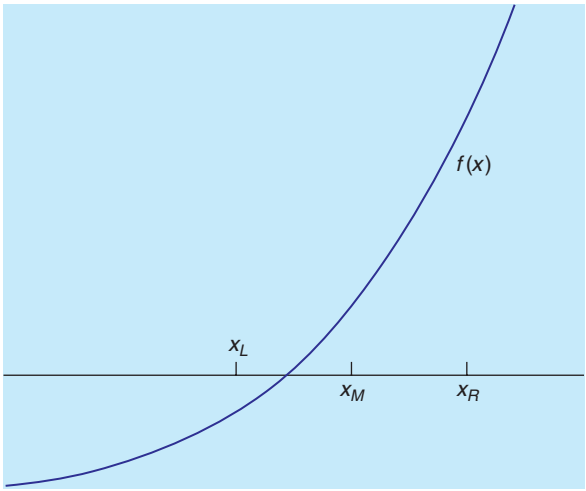


Figure 17.3 The Bisection method

root as being at the mid-point of the interval $[a, b]$. The worst that can happen is that the root is actually at x_L or x_R, in which case the error is E_1. Carrying on like this, after n bisections the worst possible error E_n is given by $E_n = |a - b|/2^n$. If we want to be sure that this is less than some specified error E, we must see to it that n satisfies the inequality $|a - b|/2^n < E$, i.e.

$$n > \frac{\log(|a - b|/E)}{\log(2)} \tag{17.2}$$

Since n is the number of bisections, it must be an integer. The smallest integer n that *exceeds* the right-hand side of Inequality (17.2) will do as the maximum number of bisections required to guarantee the given accuracy E.

The following scheme may be used to program the Bisection method. It will work for any function $f(x)$ that changes sign (in either direction) between the two values a and b, which must be found beforehand by the user.

1. Input a, b and E
2. Initialize x_L and x_R
3. Compute maximum bisections n from Inequality (17.2)
4. Repeat n times:
   Compute x_M according to Equation (17.1)
   If $f(x_L)f(x_M) > 0$ then
   Let $x_L = x_M$

otherwise
Let $x_R = x_M$

5. Display root x_M
6. Stop.

We have assumed that the procedure will not find the root exactly; the chances of this happening with real variables are infinitesimal.

The main advantage of the Bisection method is that it is guaranteed to find a root if you can find two starting values for x_L and x_R between which the function changes sign. You can also compute in advance the number of bisections needed to attain a given accuracy. Compared to Newton's method it is inefficient. Successive bisections do not necessarily move closer to the root, as usually happens with Newton's method. In fact, it is interesting to compare the two methods on the same function to see how many more steps the Bisection method requires than Newton's method. For example, to solve the equation $x^3 + x - 3 = 0$, the Bisection method takes 21 steps to reach the same accuracy as Newton's in five steps.

17.1.3 `fzero`

The MATLAB function `fzero(@f, a)` finds the zero nearest to the value `a` of the function `f` represented by the function `f.m`.

Use it to find a zero of $x^3 + x - 3$.

`fzero` doesn't appear to be able to find complex roots.

17.1.4 `roots`

The MATLAB function M-file `roots(c)` finds all the roots (zeros) of the polynomial with coefficients in the vector `c`. See `help` for details.

Use it to find a zero of $x^3 + x - 3$.

17.2 Integration

Although most 'respectable' mathematical functions can be differentiated analytically, the same cannot be said for integration. There are no general rules for

integrating, as there are for differentiating. For example, the indefinite integral of a function as simple as e^{-x^2} cannot be found analytically. We therefore need numerical methods for evaluating integrals.

This is actually quite easy, and depends on the fact that the definite integral of a function $f(x)$ between the limits $x = a$ and $x = b$ is equal to the area under $f(x)$ bounded by the x-axis and the two vertical lines $x = a$ and $x = b$. So all numerical methods for integrating simply involve more or less ingenious ways of estimating the area under $f(x)$.

17.2.1 The Trapezoidal rule

The Trapezoidal (or Trapezium) rule is fairly simple to program. The area under $f(x)$ is divided into vertical panels each of width h, called the *step-length*. If there are n such panels, then $nh = b - a$, i.e. $n = (b - a)/h$. If we join the points where successive panels cut $f(x)$, we can estimate the area under $f(x)$ as the sum of the area of the resulting trapezia (see Figure 17.4). If we call this approximation to the integral S, then

$$S = \frac{h}{2}\left[f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_i)\right], \tag{17.3}$$

where $x_i = a + ih$. Equation (17.3) is the Trapezoidal rule, and provides an estimate for the integral

$$\int_a^b f(x)\,\mathrm{d}x.$$

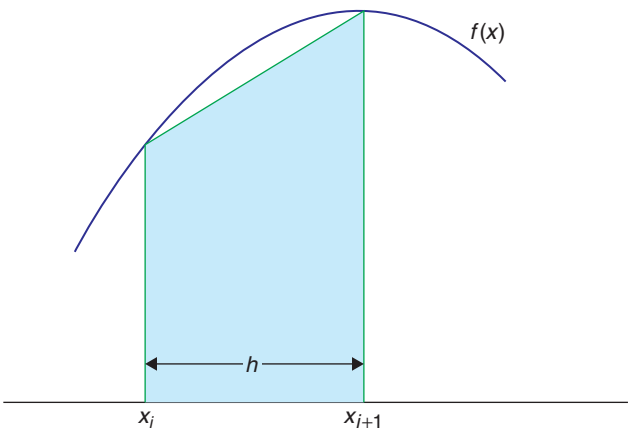


Figure 17.4 The Trapezoidal rule

Here is a function to implement the Trapezoidal rule:

```
function y = trap( fn, a, b, h )
n = (b-a)/h;
x = a + [1:n-1]*h;
y = sum(feval(fn, x));
y = h/2*(feval(fn, a) + feval(fn, b) + 2*y);
```

Note:

1. Since the summation in the rule is implemented with a vectorized formula rather than a `for` loop (to save time), the function to be integrated must use array operators where appropriate in its M-file implementation.
2. The user must choose h in such a way that the number of steps n will be an integer—a check for this could be built in.

As an exercise, integrate $f(x) = x^3$ between the limits 0 and 4 (remember to write `x.^3` in the function M-file). Call `trap` as follows:

```
s = trap(@f, 0, 4, h);
```

With $h = 0.1$, the estimate is 64.04, and with $h = 0.01$ it is 64.0004 (the exact integral is 64). You will find that as h gets smaller, the estimate gets more accurate.

This example assumes that $f(x)$ is a continuous function which may be evaluated at any x. In practice, the function could be defined at discrete points supplied as results of an experiment. For example, the speed of an object $v(t)$ might be measured every so many seconds, and one might want to estimate the distance traveled as the area under the speed-time graph. In this case, `trap` would have to be changed by replacing `fn` with a vector of function values. This is left as an exercise for the curious. Alternatively, you can use the MATLAB function `interp1` to interpolate the data. See `help`.

17.2.2 Simpson's rule

Simpson's rule is a method of numerical integration which is a good deal more accurate than the Trapezoidal rule, and should always be used before you try anything fancier. It also divides the area under the function to be integrated, $f(x)$, into vertical strips, but instead of joining the points $f(x_i)$ with straight lines, every set of three such successive points is fitted with a parabola. To ensure

that there are always an even number of panels, the step-length h is usually chosen so that there are $2n$ panels, i.e. $n = (b-a)/(2h)$.

Using the same notation as above, Simpson's rule estimates the integral as

$$S = \frac{h}{3}\left[f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_{2i}) + 4\sum_{i=1}^{n} f(x_{2i-1})\right]. \tag{17.4}$$

Coding this formula into a function M-file is left as an exercise.

If you try Simpson's rule on $f(x) = x^3$ between any limits, you will find rather surprisingly, that it gives the same result as the exact mathematical solution. This is a nice extra benefit of the rule: it integrates cubic polynomials exactly (which can be proved).

17.2.3 `quad`

Not surprisingly, MATLAB has a function `quad` to carry out numerical integration, or *quadrature* as it is also called. See `help`.

You may think there is no point in developing our own function files to handle these numerical procedures when MATLAB has its own. If you have got this far, you should be curious enough to want to know how they work, rather than treating them simply as 'black boxes'.

17.3 Numerical differentiation

The *Newton quotient* for a function $f(x)$ is given by

$$\frac{f(x+h) - f(x)}{h}, \tag{17.5}$$

where h is 'small'. As h tends to zero, this quotient approaches the first derivative, $\mathrm{d}f/\mathrm{d}x$. The Newton quotient may therefore be used to estimate a derivative numerically. It is a useful exercise to do this with a few functions for which you know the derivatives. This way you can see how small you can make h before rounding errors cause problems. Such errors arise because expression (17.5) involves subtracting two terms that eventually become equal when the limit of the computer's accuracy is reached.

As an example, the following script uses the Newton quotient to estimate $f'(x)$ for $f(x) = x^2$ (which must be supplied as a function file `f.m`) at $x = 2$, for smaller and smaller values of h (the exact answer is 4).

```
h = 1;
x = 2;
format short e
for i = 1:20
  nq = (f(x+h) - f(x))/h;
  disp( [h nq] )
  h = h / 10;
end
```

Output:

```
1            5
1.0000e-001  4.1000e+000
1.0000e-002  4.0100e+000
1.0000e-003  4.0010e+000
1.0000e-004  4.0001e+000
1.0000e-005  4.0000e+000
1.0000e-006  4.0000e+000
1.0000e-007  4.0000e+000
1.0000e-008  4.0000e+000
1.0000e-009  4.0000e+000
1.0000e-010  4.0000e+000
1.0000e-011  4.0000e+000
1.0000e-012  4.0004e+000
1.0000e-013  3.9968e+000
1.0000e-014  4.0856e+000
1.0000e-015  3.5527e+000
1.0000e-016            0
...
```

The results show that the best h for this particular problem is about 10^{-8}. But for h much smaller than this the estimate becomes totally unreliable.

Generally, the best h for a given problem can only be found by trial and error. Finding it can be a non-trivial exercise. This problem does not arise with numerical integration, because numbers are *added* to find the area, not subtracted.

17.3.1 `diff`

If `x` is a row or column vector

```
[x(1)  x(2)  ...  x(n)]
```

then the MATLAB function `diff(x)` returns a vector of differences between adjacent elements:

```
[x(2)-x(1)   x(3)-x(2)   ...   x(n)-x(n-1)]
```

The output vector is one element shorter than the input vector.

In certain problems, `diff` is helpful in finding approximate derivatives, e.g. if `x` contains displacements of an object every `h` seconds, `diff(x)/h` will be its speed.

17.4 First-order differential equations

The most interesting situations in real life that we may want to model, or represent quantitatively, are usually those in which the variables change in time (e.g. biological, electrical or mechanical systems). If the changes are continuous, the system can often be represented with equations involving the derivatives of the dependent variables. Such equations are called *differential* equations. The main aim of a lot of modeling is to be able to write down a set of differential equations (DEs) that describe the system being studied as accurately as possible. Very few DEs can be solved analytically, so once again, numerical methods are required. We will consider the simplest method of numerical solution in this section: Euler's method (Euler rhymes with 'boiler'). We also consider briefly how to improve it.

17.4.1 Euler's method

In general we want to solve a first-order DE (strictly an ordinary—ODE) of the form

$$\mathrm{d}y/\mathrm{d}x = f(x, y), \quad y(0) \text{ given.}$$

Euler's method for solving this DE numerically consists of replacing $\mathrm{d}y/\mathrm{d}x$ with its Newton quotient, so that the DE becomes

$$\frac{y(x+h) - y(x)}{h} = f(x, y).$$

After a slight rearrangement of terms, we get

$$y(x+h) = y(x) + hf(x, y). \tag{17.6}$$

Solving a DE numerically is such an important and common problem in science and engineering that it is worth introducing some general notation at this point. Suppose we want to integrate the DE over the interval $x = a$ ($a = 0$ usually) to $x = b$. We break this interval up into m steps of length h, so

$$m = (b - a)/h$$

(this is the same as the notation used in the update process of Chapter 11, except that dt used there has been replaced by the more general h here).

For consistency with MATLAB's subscript notation, if we define y_i as $y(x_i)$ (the Euler estimate at the *beginning* of step i), where $x_i = (i - 1)h$, then $y_{i+1} = y(x + h)$, at the end of step i. We can then replace Equation (17.6) by the iterative scheme

$$y_{i+1} = y_i + hf(x_i, y_i), \tag{17.7}$$

where $y_1 = y(0)$. Recall from Chapter 11 that this notation enables us to generate a vector y which we can then plot. Note also the striking similarity between Equation (17.7) and the equation in Chapter 11 representing an update process. This similarity is no coincidence. Update processes are typically modeled by DEs, and Euler's method provides an approximate solution for such DEs.

17.4.2 Example: bacteria growth

Suppose a colony of 1000 bacteria is multiplying at the rate of $r = 0.8$ per hour per individual (i.e. an individual produces an average of 0.8 offspring every hour). How many bacteria are there after 10 hours? Assuming that the colony grows continuously and without restriction, we can model this growth with the DE

$$dN/dt = rN, \quad N(0) = 1000, \tag{17.8}$$

where $N(t)$ is the population size at time t. This process is called *exponential growth*. Equation (17.8) may be solved analytically to give the well-known formula for exponential growth:

$$N(t) = N(0)e^{rt}.$$

To solve Equation (17.8) numerically, we apply Euler's algorithm to it to get

$$N_{i+1} = N_i + rhN_i, \tag{17.9}$$

where the initial value $N_1 = 1000$.

It is very easy to program Euler's method. The following script implements Equation (17.9), taking $h = 0.5$. It also computes the exact solution for comparison.

```
h = 0.5;
r = 0.8;
a = 0;
b = 10;
m = (b - a) / h;
N = zeros(1, m+1);
N(1) = 1000;
t = a:h:b;

for i = 1:m
   N(i+1) = N(i) + r * h * N(i);
end

Nex = N(1) * exp(r * t);
format bank
disp( [t' N' Nex'] )

plot(t, N ), xlabel( 'Hours' ), ylabel( 'Bacteria' )
hold on
plot(t, Nex ), hold off
```

Results are shown in Table 17.1, and also in Figure 17.5. The Euler solution is not too good. In fact, the error gets worse at each step, and after 10 hours of bacteria time it is about 72 percent. The numerical solution will improve if we make h smaller, but there will always be some value of t where the error exceeds some acceptable limit.

In some cases, Euler's method performs better than it does here, but there are other numerical methods which always do better than Euler. Two of them are discussed below. More sophisticated methods may be found in most textbooks on numerical analysis. However, Euler's method may always be used as a first approximation as long as you realize that errors may arise.

17.4.3 Alternative subscript notation

Equation 17.9 is an example of a *finite difference scheme*. The conventional finite difference notation is for the initial value to be represented by N_0, i.e. with subscript $i = 0$. N_i is then the estimate at the *end* of step i. If you want the MATLAB subscripts in the Euler solution to be the same as the finite difference subscripts, the initial value N_0 must be represented by the MATLAB scalar `N0`,

Table 17.1 Bacteria growth

Time (hours)	Euler	Predictor-Corrector	Exact
0.0	000	1000	1000
0.5	1400	1480	1492
1.0	1960	2190	2226
1.5	2744	3242	3320
2.0	3842	4798	4953
...			
5.0	28925	50422	54598
...			
8.0	217795	529892	601845
...			
10.0	836683	2542344	2980958

and you have to compute `N(1)` separately, before the `for` loop starts. You also have to display or plot the initial values separately since they will no longer be included in the MATLAB vectors `t`, `N` and `Nex` (which now have m instead of $m+1$ elements). Here is a complete script to generate the Euler solution using finite difference subscripts:

```
h = 0.5;
r = 0.8;
a = 0;
b = 10;
m = (b - a) / h;
N = zeros(1, m);      % one less element now
N0 = 1000;
N(1) = N0 + r*h*N0;  % no longer 'self-starting'

for i = 2:m
  N(i) = N(i-1) + r * h * N(i-1); %finite difference
         notation
end
```

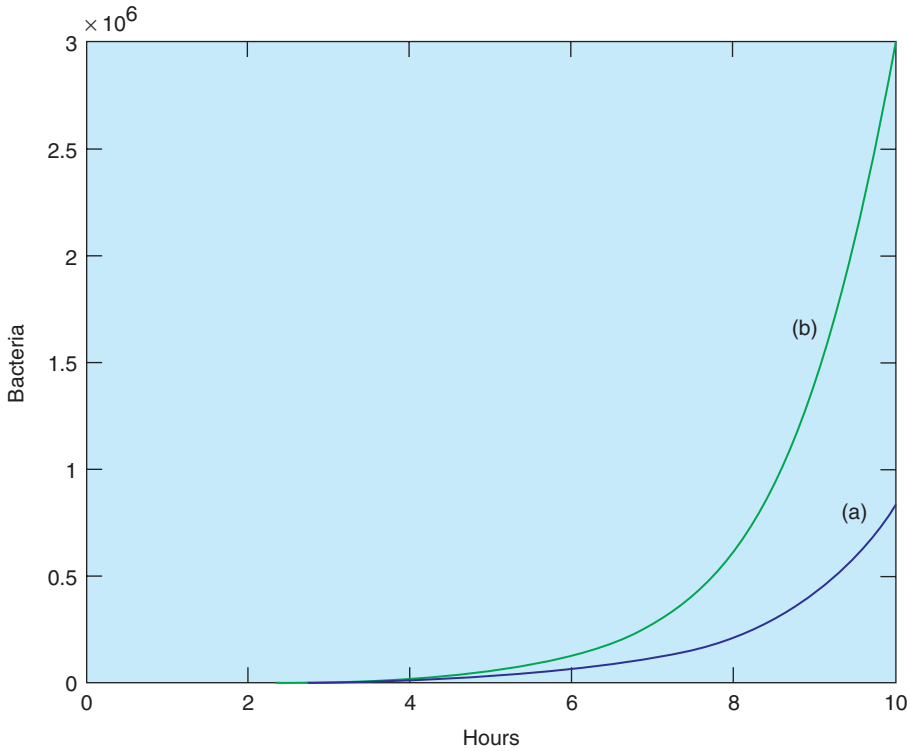


Figure 17.5 Bacteria growth: (a) Euler's method; (b) the exact solution

```
t = a+h:h:b;          % exclude initial time = a
Nex = N0 * exp(r * t);
disp( [a N0 N0] )     % display initial values separately
disp( [t' N' Nex'] )

plot(a, N0)           % plot initial values separately
hold on
plot(t, N ), xlabel( 'Hours' ), ylabel( 'Bacteria' )
plot(t, Nex ), hold off
```

17.4.4 A predictor-corrector method

One improvement on the numerical solution of the first-order DE

$$dy/dx = f(x, y), \quad y(0) \text{ given},$$

is as follows. The Euler approximation, which we are going to denote by an asterisk, is given by

$$y^*_{i+1} = y_i + hf(x_i, y_i) \tag{17.10}$$

But this formula favors the old value of y in computing $f(x_i, y_i)$ on the right-hand side. Surely it would be better to say

$$y^*_{i+1} = y_i + h[f(x_{i+1}, y^*_{i+1}) + f(x_i, y_i)]/2, \tag{17.11}$$

where $x_{i+1} = x_i + h$, since this also involves the new value y^*_{i+1} in computing f on the right-hand side? The problem of course is that y^*_{i+1} is as yet unknown, so we can't use it on the right-hand side of Equation (17.11). But we could use Euler to estimate (predict) y^*_{i+1} from Equation (17.10) and then use Equation (17.11) to correct the prediction by computing a *better* version of y^*_{i+1}, which we will call y_{i+1}. So the full procedure is:

Repeat as many times as required:

Use Euler to predict: $y^*_{i+1} = y_i + hf(x_i, y_i)$
Then correct y^*_{i+1} to: $y_{i+1} = y_i + h[f(x_{i+1}, y^*_{i+1}) + f(x_i, y_i)]/2$.

This is called a *predictor-corrector* method. The script above can easily be adapted to this problem. The relevant lines of code are:

```
for i = 1:m    % m steps of length dt
  ne(i+1) = ne(i) + r * h * ne(i);
  np = nc(i) + r * h * nc(i);
  nc(i+1) = nc(i) + r * h * (np + nc(i))/2;
  disp( [t(i+1) ne(i+1) nc(i+1) nex(i+1)] )
end;
```

`ne` stands for the 'straight' (uncorrected) Euler solution, `np` is the Euler predictor (since this is an intermediate result a vector is not needed for `np`), and `nc` is the corrector. The worst error is now only 15 percent. This is much better than the uncorrected Euler solution, although there is still room for improvement.

17.5 Linear ordinary differential equations (LODEs)

Linear ODEs with constant coefficients may be solved analytically in terms of *matrix exponentials*, which are represented in MATLAB by the function `expm`. For

an example see **Using MATLAB: Mathematics: Matrices and Linear Algebra: Matrix Powers and Exponentials**.

17.6 Runge-Kutta methods

There are a variety of algorithms, under the general name of Runge-Kutta, which can be used to integrate systems of ODEs. The formulae involved are rather complicated; they can be found in most books on numerical analysis.

However, as you may have guessed, MATLAB has plenty of ODE solvers, which are discussed in **Using MATLAB: Mathematics: Differential Equations**. Among them are `ode23` (second/third order) and `ode45` (fourth/fifth order), which implement Runge-Kutta methods. (The *order* of a numerical method is the power of h (i.e. dt) in the leading error term. Since h is generally very small, the higher the power, the smaller the error.) We will demonstrate the use of `ode23` and `ode45` here, first with a single first-order DE, and then with systems of such equations.

17.6.1 A single differential equation

Here's how to use `ode23` to solve the bacteria growth problem, Equation (17.8):

$$dN/dt = rN, \quad N(0) = 1000.$$

1. Start by writing a function file for the *right-hand side* of the DE to be solved. The function must have input variables t and N in this case (i.e. independent and dependent variables of the DE), *in that order*, e.g. create the function file **f.m** as follows:

```
function y = f(t, Nr)
y = 0.8 * Nr;
```

2. Now enter the following statements in the Command Window:

```
a = 0;
b = 10;
n0 = 1000;
[t, Nr] = ode23(@f, [a:0.5:b], n0);
```

3. Note the input arguments of `ode23`:

   `@f`: a handle for the function `f`, which contains the right-hand side of the DE;

   `[a:0.5:b]`: a vector (`tspan`) specifying the range of integration. If `tspan` has two elements (`[a b]`) the solver returns the solution evaluated at every integration step (the solver chooses the integration steps and may vary them). This form would be suitable for plotting. However, if you want to display the solution at regular time intervals, as we want to here, use the form of `tspan` with three elements as above. The solution is then returned evaluated at each time in `tspan`. The accuracy of the solution is not affected by the form of `tspan` used.

   `n0`: the initial value of the solution *N*.

4. The output arguments are two vectors: the solutions `Nr` at times `t`. After 10 hours `ode23` gives a value of 2961338 bacteria. From the exact solution in Table 17.1 we see that the error here is only 0.7 percent.

If the solutions you get from `ode23` are not accurate enough, you can request greater accuracy with an additional optional argument. See `help`.

If you need still more accurate numerical solutions, you can use `ode45` instead. It gives a final value for the bacteria of 2981290—an error of about 0.01%.

17.6.2 Systems of differential equations: chaos

The reason that weather prediction is so difficult and forecasts are so erratic is no longer thought to be the complexity of the system but the nature of the DEs modeling it. These DEs belong to a class referred to as *chaotic*. Such equations will produce wildly different results when their initial conditions are changed infinitesimally. In other words, accurate weather prediction depends crucially on the accuracy of the measurements of the initial conditions.

Edward Lorenz, a research meteorologist, discovered this phenomenon in 1961. Although his original equations are far too complex to consider here, the following much simpler system has the same essential chaotic features:

$$dx/dt = 10(y - x), \tag{17.12}$$

$$dy/dt = -xz + 28x - y, \tag{17.13}$$

$$dz/dt = xy - 8z/3. \tag{17.14}$$

This system of DEs may be solved very easily with the MATLAB ODE solvers. The idea is to solve the DEs with certain initial conditions, plot the solution, then change the initial conditions very slightly, and superimpose the new solution over the old one to see how much it has changed.

We begin by solving the system with the initial conditions $x(0) = -2$, $y(0) = -3.5$ and $z(0) = 21$.

1. Write a function file `lorenz.m` to represent the right-hand sides of the system as follows:

```
function f = lorenz(t, x)
f = zeros(3,1);
f(1) = 10 * (x(2) - x(1));
f(2) = -x(1) * x(3) + 28 * x(1) - x(2);
f(3) = x(1) * x(2) - 8 * x(3) / 3;
```

   The three elements of the MATLAB vector `x`, i.e. `x(1)`, `x(2)` and `x(3)`, represent the three dependent scalar variables x, y and z respectively. The elements of the vector `f` represent the right-hand sides of the three DEs. When a vector is returned by such a DE function it must be a column vector, hence the statement

```
f = zeros(3,1);
```

2. Now use the following commands to solve the system from $t = 0$ to $t = 10$, say:

```
x0 = [-2 -3.5 21];   % initial values in a vector
[t, x] = ode45(@lorenz, [0 10], x0);
plot(t,x)
```

   Note that we are use `ode45` now, since it is more accurate. (If you aren't using a Pentium it will take quite a few seconds for the integration to be completed.)

   You will see three graphs, for x, y and z (in different colors).

3. It's easier to see the effect of changing the initial values if there is only one graph in the figure to start with. It is in fact best to plot the solution $y(t)$ on its own.

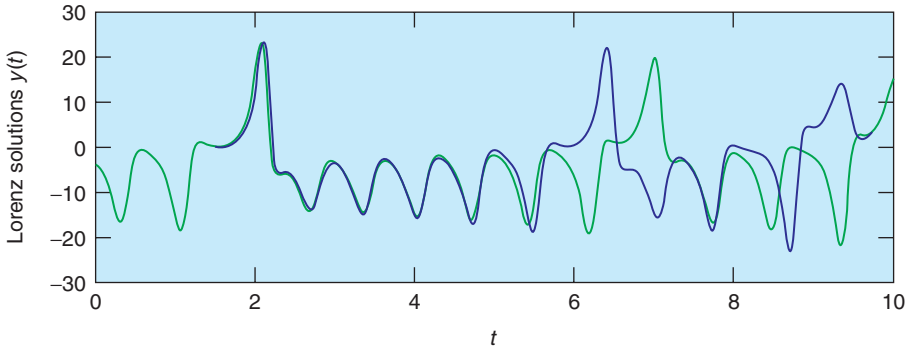


Figure 17.6 Chaos?

The MATLAB solution `x` is actually a matrix with three columns (as you can see from `whos`). The solution $y(t)$ that we want will be the second column, so to plot it by itself use the command

```
plot(t,x(:,2),'g')
```

Then keep the graph on the axes with the command `hold`.

Now we can see the effect of changing the initial values. Let's just change the initial value of $x(0)$, from -2 to -2.04—that's a change of only 2 percent, and in only one of the three initial values. The following commands will do this, solve the DEs, and plot the new graph of $y(t)$ (in a different color):

```
x0 = [-2.04 -3.5 21];
[t, x] = ode45(@lorenz, [0 10], x0);
plot(t,x(:,2),'r')
```

You should see (Figure 17.6) that the two graphs are practically indistinguishable until t is about 1.5. The discrepancy grows quite gradually, until t reaches about 6, when the solutions suddenly and shockingly flip over in opposite directions. As t increases further, the new solution bears no resemblance to the old one.

Now solve the system (17.12)–(17.14) with the original initial values using `ode23` this time:

```
x0 = [-2 -3.5 21];
[t,x] = ode23(@lorenz, [0 10], x0);
```

Plot the graph of $y(t)$ only—`x(:,2)`—and then superimpose the `ode45` solution *with the same initial values* (in a different color).

A strange thing happens—the solutions begin to deviate wildly for $t > 1.5$! The initial conditions are the same—the only difference is the order of the Runge-Kutta method.

Finally solve the system with `ode23s` and superimpose the solution. (The `s` stands for 'stiff'. For a stiff DE, solutions can change on a time scale that is very short compared to the interval of integration.) The `ode45` and `ode23s` solutions only start to diverge at $t > 5$.

The explanation is that `ode23`, `ode23s` and `ode45` all have numerical inaccuracies (if one could compare them with the exact solution—which incidentally can't be found). However, the numerical inaccuracies are *different* in the three cases. This difference has the same effect as starting the numerical solution with very slightly different initial values.

How do we ever know when we have the 'right' numerical solution? Well, we don't—the best we can do is increase the accuracy of the numerical method until no further wild changes occur over the interval of interest. So in our example we can only be pretty sure of the solution for $t < 5$ (using `ode23s` or `ode45`). If that's not good enough, you have to find a more accurate DE solver.

So beware: 'chaotic' DEs are very tricky to solve!

Incidentally, if you want to see the famous 'butterfly' picture of chaos, just plot x against z as time increases (the resulting graph is called a *phase plane* plot). The following command will do the trick:

```
plot(x(:,1), x(:,3))
```

What you will see is a static 2-D projection of the *trajectory*, i.e. the solution developing in time. **Demos** in the MATLAB Launch Pad include an example which enables you to see the trajectory evolving dynamically in 3-D (**Demos: Graphics: Lorenz attractor animation**).

17.6.3 Passing additional parameters to an ODE solver

In the above examples of the MATLAB ODE solvers the coefficients in the right-hand sides of the DEs (e.g. the value 28 in Equation (17.13)) have all been

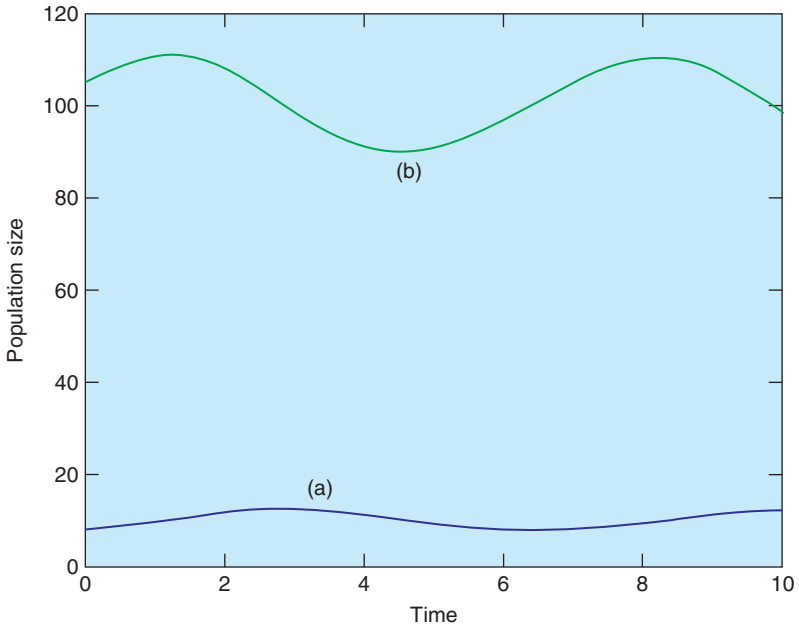


Figure 17.7 Lotka-Volterra model: (a) predator; (b) prey

constants. In a real modeling situation, you will most likely want to change such coefficients frequently. To avoid having to edit the function files each time you want to change a coefficient, you can pass the coefficients as additional parameters to the ODE solver, which in turn passes them to the DE function. To see how this may be done, consider the Lotka-Volterra *predator-prey* model:

$$dx/dt = px - qxy \tag{17.15}$$

$$dy/dt = rxy - sy, \tag{17.16}$$

where $x(t)$ and $y(t)$ are the prey and predator population sizes at time t, and p, q, r and s are biologically determined parameters. For this example, we take $p = 0.4$, $q = 0.04$, $r = 0.02$, $s = 2$, $x(0) = 105$ and $y(0) = 8$.

First, write a function M-file, `volterra.m` as follows:

```
function f = volterra(t, x, p, q, r, s)
f = zeros(2,1);
f(1) = p*x(1) - q*x(1)*x(2);
f(2) = r*x(1)*x(2) - s*x(2);
```

Then enter the following statements in the Command Window, which generate the characteristically oscillating graphs in Figure 17.7:

```
p = 0.4; q = 0.04; r = 0.02; s = 2;
[t,x] = ode23(@volterra,[0 10],[105; 8],[],p,q,r,s);
plot(t, x)
```

Note:

- The additional parameters (`p`, `q`, `r` and `s`) have to follow the fourth input argument (`options`—see `help`) of the ODE solver. If no options have been set (as in our case), use `[]` as a placeholder for the `options` parameter.

You can now change the coefficients from the Command Window and get a new solution, without editing the function file.

17.7 A partial differential equation

The numerical solution of partial differential equations (PDEs) is a vast subject, which is beyond the scope of this book. However, a class of PDEs called *parabolic* often lead to solutions in terms of sparse matrices, which were mentioned briefly in Chapter 16. One such example is considered in this section.

17.7.1 Heat conduction

The conduction of heat along a thin uniform rod may be modeled by the partial differential equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \tag{17.17}$$

where $u(x, t)$ is the temperature distribution a distance x from one end of the rod at time t, and assuming that no heat is lost from the rod along its length.

Half the battle in solving PDEs is mastering the notation. We set up a rectangular grid, with step-lengths of h and k in the x and t directions respectively. A general point on the grid has coordinates $x_i = ih$, $y_j = jk$. A concise notation for $u(x, t)$ at x_i, y_j is then simply $u_{i,j}$.

Truncated Taylor series may then be used to approximate the PDE by a *finite difference scheme*. The left-hand side of Equation (17.17) is usually approximated by a *forward difference*:

$$\frac{\partial u}{\partial t} = \frac{u_{i,j+1} - u_{i,j}}{k}$$

One way of approximating the right-hand side of Equation (17.17) is by the scheme

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}. \tag{17.18}$$

This leads to a scheme, which although easy to compute, is only conditionally stable.

If however we replace the right-hand side of the scheme in Equation (17.18) by the mean of the finite difference approximation on the jth and $(j+1)$th time rows, we get (after a certain amount of algebra!) the following scheme for Equation (17.17):

$$-ru_{i-1,j+1} + (2+2r)u_{i,j+1} - ru_{i+1,j+1} = ru_{i-1,j} + (2-2r)u_{i,j} + ru_{i+1,j}, \tag{17.19}$$

where $r = k/h^2$. This is known as the Crank-Nicolson *implicit* method, since it involves the solution of a system of simultaneous equations, as we shall see.

To illustrate the method numerically, let's suppose that the rod has a length of 1 unit, and that its ends are in contact with blocks of ice, i.e. the *boundary conditions* are

$$u(0,t) = u(1,t) = 0. \tag{17.20}$$

Suppose also that the initial temperature (*initial condition*) is

$$u(x,0) = \begin{cases} 2x, & 0 \le x \le 1/2, \\ 2(1-x), & 1/2 \le x \le 1. \end{cases} \tag{17.21}$$

(This situation could come about by heating the center of the rod for a long time, with the ends kept in contact with the ice, removing the heat source at time $t = 0$.) This particular problem has symmetry about the line $x = 1/2$; we exploit this now in finding the solution.

If we take $h = 0.1$ and $k = 0.01$, we will have $r = 1$, and Equation (17.19) becomes

$$-u_{i-1,j+1} + 4u_{i,j+1} - u_{i+1,j+1} = u_{i-1,j} + u_{i+1,j}. \tag{17.22}$$

Putting $j = 0$ in Equation (17.22) generates the following set of equations for the unknowns $u_{i,1}$ (i.e. after one time step k) up to the mid-point of the rod, which is represented by $i = 5$, i.e. $x = ih = 0.5$. The subscript $j = 1$ has been

dropped for clarity:

$$\begin{aligned} 0 + 4u_1 - u_2 &= 0 + 0.4 \\ -u_1 + 4u_2 - u_3 &= 0.2 + 0.6 \\ -u_2 + 4u_3 - u_4 &= 0.4 + 0.8 \\ -u_3 + 4u_4 - u_5 &= 0.6 + 1.0 \\ -u_4 + 4u_5 - u_6 &= 0.8 + 0.8. \end{aligned}$$

Symmetry then allows us to replace u_6 in the last equation by u_4. These equations can be written in matrix form as

$$\begin{bmatrix} 4 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & -2 & 4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 0.8 \\ 1.2 \\ 1.6 \\ 1.6 \end{bmatrix}. \tag{17.23}$$

The matrix (**A**) on the left of Equations (17.23) is known as a *tridiagonal* matrix. Having solved for the $u_{i,1}$ we can then put $j = 1$ in Equation (17.22) and proceed to solve for the $u_{i,2}$, and so on. The system (17.23) can of course be solved directly in MATLAB with the left division operator. In the script below, the general form of Equations (17.23) is taken as

$$\mathbf{Av} = \mathbf{g}. \tag{17.24}$$

Care needs to be taken when constructing the matrix **A**. The following notation is often used:

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & & \cdots & & & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} & \\ & & & & a_n & b_n & \end{bmatrix}.$$

A is an example of a sparse matrix (see Chapter 16).

The script below implements the general Crank-Nicolson scheme of Equation (17.19) to solve this particular problem over 10 time steps of $k = 0.01$. The step-length is specified by $h = 1/(2n)$ because of symmetry. r is therefore not restricted to the value 1, although it takes this value here. The script exploits the sparsity of **A** by using the `sparse` function.

```
format compact
n = 5;
k = 0.01;
h =  1 / (2 * n);                          % symmetry assumed
r = k / h ^ 2;

% set up the (sparse) matrix A
b = sparse(1:n, 1:n, 2+2*r, n, n);         % b(1) .. b(n)
c = sparse(1:n-1, 2:n, -r, n, n);          % c(1) .. c(n-1)
a = sparse(2:n, 1:n-1, -r, n, n);          % a(2) ..
A = a + b + c;
A(n, n-1) = -2 * r;                        % symmetry: a(n)
full(A)                                    %
disp(' ')

u0 = 0;                      % boundary condition (Eq 19.20)
u = 2*h*[1:n]                % initial conditions (Eq 19.21)
u(n+1) = u(n-1);             % symmetry
disp([0 u(1:n)])

for t = k*[1:10]
  g = r * ([u0 u(1:n-1)] + u(2:n+1)) ...
                         + (2 - 2 * r) * u(1:n);
                            % Eq 19.19
  v = A\g';                 % Eq 19.24
  disp([t v'])
  u(1:n) = v;
  u(n+1) = u(n-1);           % symmetry
end
```

Note:

- to preserve consistency between the formal subscripts of Equation (17.19) etc. and MATLAB subscripts, u_0 (the boundary value) is represented by the scalar `u0`.

In the following output the first column is time, and subsequent columns are the solutions at intervals of h along the rod:

```
     0    0.2000    0.4000    0.6000    0.8000    1.0000
0.0100    0.1989    0.3956    0.5834    0.7381    0.7691
0.0200    0.1936    0.3789    0.5397    0.6461    0.6921
   ...
0.1000    0.0948    0.1803    0.2482    0.2918    0.3069
```

MATLAB has some built-in PDE solvers. See **Using MATLAB: Mathematics: Differential Equations: Partial Differential Equations**.

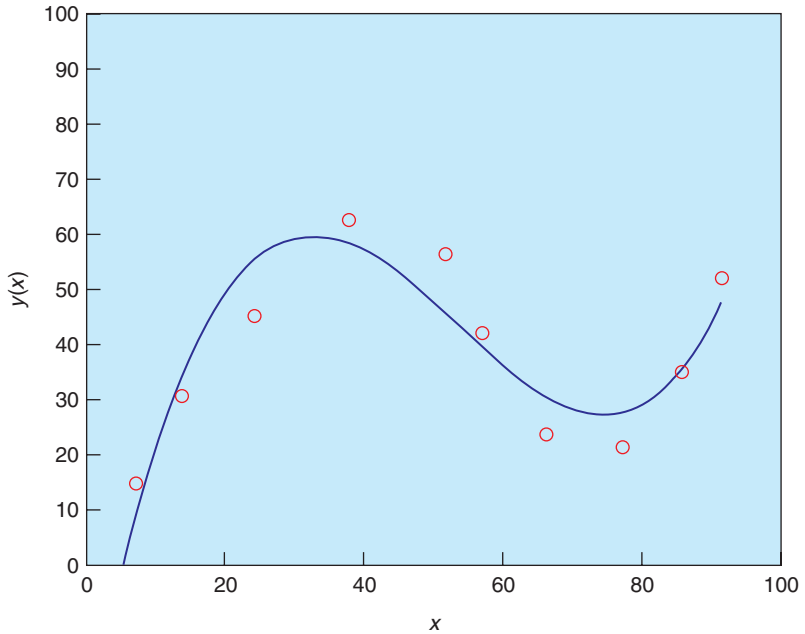


Figure 17.8 A cubic polynomial fit

17.8 Other numerical methods

The ODEs considered earlier in this chapter are all *initial value* problems. For *boundary value* problem solvers, see **Using MATLAB: Mathematics: Differential Equations: Boundary Value Problems for ODEs**.

MATLAB has a large number of functions for handling other numerical procedures, such as curve fitting, correlation, interpolation, minimization, filtering and convolution, and (fast) Fourier transforms. Consult **Using MATLAB: Mathematics: Polynomials and Interpolation** and **Data Analysis and Statistics**.

Here's an example of curve fitting. The following script enables you to plot data points interactively. When you have finished plotting points (signified when the x coordinates of your last two points differ by less than 2 in absolute value) a cubic polynomial is fitted and drawn (see Figure 17.8).

```
% Interactive script to fit a cubic to data points

clf
hold on
axis([0 100 0 100]);
```

```
diff = 10;
xold = 68;
i = 0;
xp = zeros(1);        % data points
yp = zeros(1);
while diff > 2
[a b] = ginput(1);
diff = abs(a - xold);
if diff > 2
    i = i + 1;
    xp(i) = a;
    yp(i) = b;
    xold = a;
    plot(a, b, 'ok')
  end
end

p = polyfit(xp, yp, 3 );
x = 0:0.1:xp(length(xp));
y= p(1)*x.^3 + p(2)*x.^2 + p(3)*x + p(4);
plot(x,y), title( 'cubic polynomial fit'), ...
   ylabel('y(x)'), xlabel('x')
hold off
```

Polynomial fitting may also be done interactively in a figure window, with **Tools -> Basic Fitting**.

Summary

- A numerical method is an approximate computer method for solving a mathematical problem which often has no analytical solution.
- A numerical method is subject to two distinct types of error: rounding error in the computer solution, and *truncation error*, where an infinite mathematical process, like taking a limit, is approximated by a finite process.
- MATLAB has a large number of useful functions for handling numerical methods.

EXERCISES

17.1 Use Newton's method in a script to solve the following (you may have to experiment a bit with the starting values). Check all your answers with `fzero`. Check the answers involving polynomial equations with `roots`.

Hint: Use `fplot` to get an idea of where the roots are, e.g.

```
fplot('x^3-8*x^2+17*x-10', [0 3])
```

The Zoom feature also helps. In the figure window select the Zoom In button (magnifying glass) and click on the part of the graph you want to magnify.

(a) $x^4 - x = 10$ (two real roots and two complex roots)
(b) $e^{-x} = \sin x$ (infinitely many roots)
(c) $x^3 - 8x^2 + 17x - 10 = 0$ (three real roots)
(d) $\log x = \cos x$
(e) $x^4 - 5x^3 - 12x^2 + 76x - 79 = 0$ (four real roots)

17.2 Use the Bisection method to find the square root of 2, taking 1 and 2 as initial values of x_L and x_R. Continue bisecting until the maximum error is less than 0.05 (use Inequality (17.2) of Section 17.1 to determine how many bisections are needed).

17.3 Use the Trapezoidal rule to evaluate $\int_0^4 x^2 dx$, using a step-length of $h = 1$.

17.4 A human population of 1000 at time $t = 0$ grows at a rate given by

$$dN/dt = aN,$$

where $a = 0.025$ per person per year. Use Euler's method to project the population over the next 30 years, working in steps of (a) $h = 2$ years, (b) $h = 1$ year and (c) $h = 0.5$ years. Compare your answers with the exact mathematical solution.

17.5 Write a function file `euler.m` which starts with the line

```
function [t, n] = euler(a, b, dt)
```

and which uses Euler's method to solve the bacteria growth DE (17.8). Use it in a script to compare the Euler solutions for $dt = 0.5$ and 0.05 with the exact solution. Try to get your output looking like this:

```
time  dt = 0.5  dt = 0.05  exact

   0  1000.00   1000.00    1000.00
0.50  1400.00   1480.24    1491.82
1.00  1960.00   2191.12    2225.54
...
5.00  28925.47  50504.95   54598.15
```

17.6 The basic equation for modeling radioactive decay is

$$dx/dt = -rx,$$

where x is the amount of the radioactive substance at time t, and r is the decay rate.

Some radioactive substances decay into other radioactive substances, which in turn also decay. For example, Strontium 92 ($r_1 = 0.256$ per hr) decays into Yttrium 92 ($r_2 = 0.127$ per hr), which in turn decays into Zirconium. Write down a pair of differential equations for Strontium and Yttrium to describe what is happening.

Starting at $t = 0$ with 5×10^{26} atoms of Strontium 92 and none of Yttrium, use the Runge-Kutta method (`ode23`) to solve the equations up to $t = 8$ hours in steps of 1/3 hr. Also use Euler's method for the same problem, and compare your results.

17.7 The springbok (a species of small buck, not rugby players!) population $x(t)$ in the Kruger National Park in South Africa may be modeled by the equation

$$dx/dt = (r - bx \sin at)x,$$

where r, b, and a are constants. Write a program which reads values for r, b, and a, and initial values for x and t, and which uses Euler's method to compute the impala population at monthly intervals over a period of two years.

17.8 The luminous efficiency (ratio of the energy in the visible spectrum to the total energy) of a black body radiator may be expressed as a percentage by the formula

$$E = 64.77T^{-4} \int_{4\times10^{-5}}^{7\times10^{-5}} x^{-5}(e^{1.432/Tx} - 1)^{-1} dx,$$

where T is the absolute temperature in degrees Kelvin, x is the wavelength in cm, and the range of integration is over the visible spectrum.

Write a general function `simp(fn, a, b, h)` to implement Simpson's rule as given in Equation (17.4).

Taking $T = 3500°$K, use `simp` to compute E, firstly with 10 intervals ($n = 5$), and then with 20 intervals ($n = 10$), and compare your results.

(Answers: 14.512725% for $n = 5$; 14.512667% for $n = 10$)

17.9 Van der Pol's equation is a second-order nonlinear differential equation which may be expressed as two first-order equations as follows:

$$dx_1/dt = x_2$$
$$dx_2/dt = \epsilon(1 - x_1^2)x_2 - b^2 x_1.$$

The solution of this system has a stable limit cycle, which means that if you plot the phase trajectory of the solution (the plot of x_1 against x_2) starting at any point in the positive x_1–x_2 plane, it always moves continuously into the same

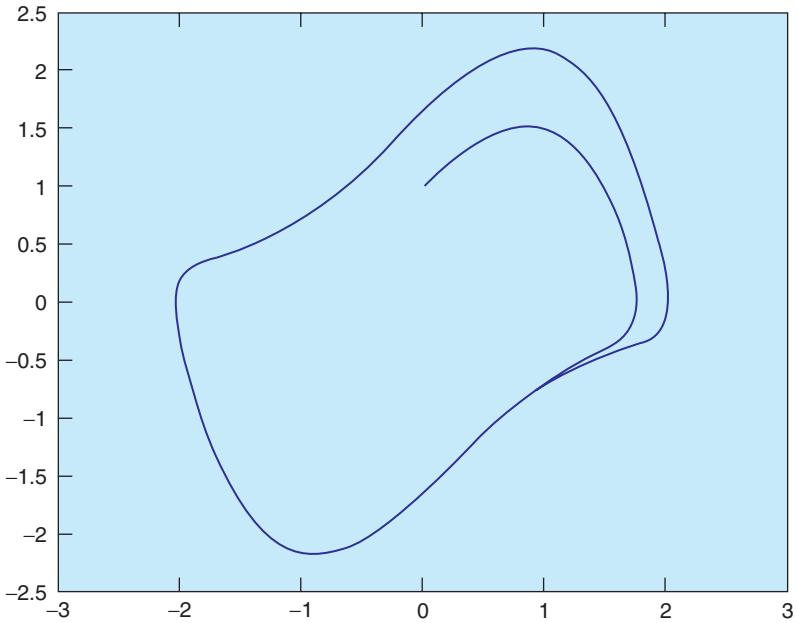


Figure 17.9 A trajectory of Van der Pol's equation

closed loop. Use `ode23` to solve this system numerically, for $x_1(0) = 0$, and $x_2(0) = 1$. Draw some phase trajectories for $b = 1$ and ϵ ranging between 0.01 and 1.0. Figure 17.9 shows you what to expect.

Appendix A
Syntax quick reference

This appendix gives examples of the most commonly used MATLAB syntax in this book.

A.1 Expressions

```
x = 2 ^ (2 * 3) / 4;
x = A \ b;              % solution of linear equations
a == 0 & b < 0          % a equals 0 AND b less than 0
a ~= 4 | b > 0          % a not equal to 4 OR b greater than 0
```

A.2 Function M-files

```
function y = f(x)                                % save as f.m
% comment for help

function [out1, out2] = plonk(in1, in2, in3)  % save as plonk.m
% Three input arguments, two outputs
  ...

function junk       % no input/output arguments; save as junk.m

[t, x] = ode45(@lorenz, [0 10], x0);   % function handle with @
```

A.3 Graphics

```
plot(x, y), grid     % plots vector y against vector x on a grid

plot(x, y, 'b--')    % plots a blue dashed line
```

```
plot(x, y, 'go')     % plots green circles

plot(y)  % if y is a vector plots elements against row numbers
         % if y is a matrix, plots columns against row numbers

plot(x1, y1, x2, y2) % plots y1 against x1 and
                       y2 against x2 on same graph

semilogy(x, y)    % uses a log10 scale for y

polar(theta, r)   % generates a polar plot
```

A.4 `if` and `switch`

```
if condition
  statement         % executed if condition true
end;

if condition
  statement1        % executed if condition true
else
  statement2        % executed if condition false
end;

if a == 0           % test for equality
  x = -c / b;
else
  x = -b / (2*a);
end;

if condition1       % jumps off ladder at first true condition
  statement1
elseif condition2   % elseif one word!
  statement2
elseif condition3
  statement3
...
else
  statementE
end;

if condition statement1, else statement2, end   % command line
```

```
switch lower(expr)            % expr is string or scalar
  case {'linear','bilinear'}
    disp('Method is linear')
  case 'cubic'
    disp('Method is cubic')
  case 'nearest'
    disp('Method is nearest')
  otherwise
    disp('Unknown method.')
end
```

A.5 for and while

```
for i = 1:n      % repeats statements n times
  statements
end;

for i = 1:3:8    % i takes values 1, 4, 7
  ...
end;

for i = 5:-2:0   % i takes values 5, 3, 1
  ...
end;

for i = v        % index i takes on each element of vector v
  statements
end;

for v = a        % index v takes on each column of matrix a
  statements
end;

for i = 1:n, statements, end      % command line version

try,
  statements,
catch,
  statements,
end
while condition   % repeats statements while condition is true
  statements
end;

while condition statements, end   % command line version
```

A.6 Input/output

```
disp( x )

disp( 'Hello there' )

disp([a b])     % two scalars on one line

disp([x' y'])   % two columns (vectors x and y must be same length)

disp( ['The answer is ', num2str(x)] )

fprintf( '\n' )                 %  new line

fprintf( '%5.1f\n', 1.23 )      %  **1.2

fprintf( '%12.2e\n', 0.123 )    %  ***1.23e-001

fprintf( '%4.0f and %7.2f\n', 12.34, -5.6789 )
                                %  **12 and **-5.68

fprintf( 'Answers are: %g %g\n', x, y )  % matlab decides on format

fprintf( '%10s\n', str )       % left-justified string

x = input( 'Enter value of x: ' )

name = input( 'Enter your name without apostrophes: ', 's' )
```

A.7 load/save

```
load filename           % retrieves all variables
                          from binary file filename.mat
load x.dat              % imports matrix x from ASCII file x.dat
save filename x y z     % saves x y and z in filename.mat
save                    % saves all workspace variables in matlab.mat
save filename x /ascii  % saves x in filename
                          (as ASCII file)
```

A.8 Vectors and matrices

```
a(3,:)                  % third row

a(:,2)                  % second column
```

```
v(1:2:9)                  % every second element from 1 to 9

v([2 4 5]) = [ ]          % removes second, fourth and fifth elements

v(logical([0 1 0 1 0]))   % second and fourth elements only

v'                        % transpose
```

Appendix B
Operators

Table B.1 Operator precedence (see **Help** on `operator precedence`)

Precedence	Operators
1.	`( )`
2.	`^ .^ ' .'` (pure transpose)
3.	`+` (unary plus) `-` (unary minus) `~` (NOT)
4.	`* / \ .* ./ .\`
5.	`+` (addition) `-` (subtraction)
6.	`:`
7.	`> < >= <= == ~=`
8.	`&` (AND)
9.	`|` (OR)

Appendix C
Command and functionquick reference

This appendix is not exhaustive; it lists most of the MATLAB commands and functions used in the text, as well as a few more.

For a complete list by category (with links to detailed descriptions) see the online documentation **MATLAB: Reference: MATLAB Function Reference: Functions by Category**.

The command `help` by itself displays a list of all the function categories (each in its own directory):

```
matlab\general        -  General purpose commands.
matlab\ops            -  Operators and special characters.
matlab\lang           -  Programming language constructs.
matlab\elmat          -  Elementary matrices and matrix
                         manipulation.
matlab\elfun          -  Elementary math functions.
matlab\specfun        -  Specialized math functions.
matlab\matfun         -  Matrix functions - numerical
                         linear algebra.
matlab\datafun        -  Data analysis and Fourier
                         transforms.
matlab\audio          -  Audio support.
matlab\polyfun        -  Interpolation and polynomials.
matlab\funfun         -  Function functions and ODE
                         solvers.
matlab\sparfun        -  Sparse matrices.
matlab\graph2d        -  Two dimensional graphs.
matlab\graph3d        -  Three dimensional graphs.
matlab\specgraph      -  Specialized graphs.
matlab\graphics       -  Handle Graphics.
matlab\uitools        -  Graphical user interface tools.
matlab\strfun         -  Character strings.
```

```
matlab\iofun          -  File input/output.
matlab\timefun        -  Time and dates.
matlab\datatypes      -  Data types and structures.
matlab\verctrl        -  Version control.
matlab\winfun         -  Windows Operating System
matlab\DDE/ActiveX)   -  Interface Files
matlab\demos          -  Examples and demonstrations.
toolbox\local         -  Preferences.
MATLABR12\work        -  (No table of contents file)

For more help on directory/topic, type "help topic".
```

C.1 General purpose commands

C.1.1 Managing commands

`demo`	Run demos
`help`	Online help
`helpwin`	Display categories of functions with links to each category
`lookfor`	Keyword search through `help` entries
`type`	List M-file
`what`	Directory listing of M- and MAT-files
`which`	Locate functions and files

C.1.2 Managing variables and the workspace

`clear`	Clear variables and functions from memory
`disp`	Display matrix or text
`length`	Length of a vector
`load`	Retrieve variables from disk
`save`	Save workspace variables to disk
`size`	Array dimensions
`who, whos`	List variables in the workspace

C.1.3 Files and the operating system

`beep`	Produce a beep sound
`cd`	Change current working directory

`delete`	Delete file
`diary`	Save text of MATLAB session
`dir`	Directory listing
`edit`	Edit an M-file
`!`	Execute operating system command

C.1.4 Controlling the Command Window

`clc`	Clear Command Window
`echo`	Echo commands in script
`format`	Set output format for `disp`
`home`	Send cursor home
`more`	Control paged output

C.1.5 Starting and quitting MATLAB

`exit`	Terminate MATLAB
`quit`	Terminate MATLAB
`startup`	M-file executed when MATLAB starts

C.2 Logical functions

`all`	True if all elements of vector are true (non-zero)
`any`	True if any element of vector is true
`exist`	Check if variable or file exists
`find`	Find indices of non-zero elements
`is*`	Detect various states
`logical`	Convert numeric values to logical

C.3 Language constructs and debugging

C.3.1 MATLAB as a programming language

`error`	Display error message
`eval`	Interpret string containing MATLAB expression

`feval`	Function evaluation
`for`	Repeat statements a specific number of times
`global`	Define global variable
`if`	Conditionally execute statements
`persistent`	Define persistent variable
`switch`	Switch among several cases
`try`	Begin `try` block
`while`	Repeat statements conditionally

C.3.2 Interactive input

`input`	Prompt user for input
`keyboard`	Invoke keyboard as a script file
`menu`	Generate menu of choices for user input
`pause`	Wait for user response

C.4 Matrices and matrix manipulation

C.4.1 Elementary matrices

`eye`	Identity matrix
`linspace`	Vector with linearly spaced elements
`ones`	Matrix of ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`zeros`	Matrix of zeros
`:` (colon)	Vector with regularly spaced elements

C.4.2 Special variables and constants

`ans`	Most recent answer
`eps`	Floating point relative accuracy
`i or j`	$\sqrt{-1}$
`Inf`	Infinity
`NaN`	Not-a-Number
`nargin, nargout`	Number of actual function arguments
`pi`	3.14159 26535 897 …
`realmax`	Largest positive floating point number
`realmin`	Smallest positive floating point number
`varargin, varargout`	Pass or return variable numbers of arguments

C.4.3 Time and date

`calendar`	Calendar
`clock`	Wall clock (complete date and time)
`date`	You'd never guess
`etime`	Elapsed time
`tic, toc`	Stopwatch
`weekday`	Day of the week

C.4.4 Matrix manipulation

`cat`	Concatenate arrays
`diag`	Create or extract diagonal
`fliplr`	Flip in left/right direction
`flipud`	Flip in up/down direction
`repmat`	Replicate and tile an array
`reshape`	Change shape
`rot90`	Rotate 90°
`tril`	Extract lower tridiagonal part
`triu`	Extract upper tridiagonal part

C.4.5 Specialized matrices

`gallery`	Test matrices
`hilb`	Hilbert matrix
`magic`	Magic square
`pascal`	Pascal matrix
`wilkinson`	Wilkinson's eigenvalue test matrix

C.5 Mathematical functions

`abs`	Absolute value
`acos, acosh`	Inverse cosine and inverse hyperbolic cosine
`acot, acoth`	Inverse cotangent and inverse hyperbolic cotangent
`acsc, acsch`	Inverse cosecant and inverse hyperbolic cosecant
`angle`	Phase angle
`asec, asech`	Inverse secant and inverse hyperbolic secant

`asin, asinh`	Inverse sine and inverse hyperbolic sine
`atan, atanh`	Inverse tangent (two quadrant) and inverse hyperbolic tangent
`atan2`	Inverse tangent (four quadrant)
`bessel`	Bessel function
`ceil`	Round up
`conj`	Complex conjugate
`cos, cosh`	Cosine and hyperbolic cosine
`cot, coth`	Cotangent and hyperbolic cotangent
`csc, csch`	Cosecant and hyperbolic cosecant
`erf`	Error function
`exp`	Exponential
`fix`	Round toward zero
`floor`	Round down
`gamma`	Gamma function
`imag`	Imaginary part
`log`	Natural logarithm
`log2`	Dissect floating point numbers into exponent and mantissa
`log10`	Common logarithm
`mod`	Modulus (signed remainder after division)
`rat`	Rational approximation
`real`	Real part
`rem`	Remainder after division
`round`	Round toward nearest integer
`sec, sech`	Secant and hyperbolic secant
`sign`	Signum function
`sin, sinh`	Sine and hyperbolic sine
`sqrt`	Square root =-16pt =8pt
`tan, tanh`	Tangent and hyperbolic tangent

C.6 Matrix functions

`det`	Determinant
`eig`	Eigenvalues and eigenvectors
`expm`	Matrix exponential
`inv`	Matrix inverse
`poly`	Characteristic polynomial
`rank`	Number of linearly independent rows or columns
`rcond`	Condition estimator
`trace`	Sum of diagonal elements
`{}\` and `/`	Linear equation solution

C.7 Data analysis

`cumprod`	Cumulative product
`cumsum`	Cumulative sum
`diff`	Difference function
`fft`	One-dimensional fast Fourier transform
`max`	Largest element
`mean`	Average value of elements
`median`	Median value of elements
`min`	Smallest element
`prod`	Product of elements
`sort`	Sort into ascending order
`std`	Standard deviation
`sum`	Sum of elements
`trapz`	Trapezoidal rule for numerical integration

C.8 Polynomial functions

`polyfit`	Fit polynomial to data
`polyval`	Evaluate polynomial
`roots`	Find polynomial roots

C.9 Function functions

`bvp4c`	Solve two-point boundary value problems for ODEs
`fmin`	Minimize function of one variable
`fmins`	Minimize function of several variables
`fzero`	Find zero of function of one variable
`ode23`, `ode23s`, `ode45`	Solve initial value problems for ODEs
`quad`	Numerical integration

C.10 Sparse matrix functions

`full`	Convert sparse matrix to full matrix
`sparse`	Construct sparse matrix from non-zeros and subscripts
`spy`	Visualize sparse matrix

C.11 Character string functions

`char`	characters from ASCII codes
`double`	ASCII codes of characters
`lower`	Convert string to lower case
`sprintf`	Write formatted data to a string
`str2mat`	String to matrix conversion
`strcat`	String concatenation
`strcmp`	Compare strings
`upper`	Convert string to upper case

C.12 File I/O functions

`fclose`	Close one or more open files
`feof`	Test for end-of-file
`fopen`	Open file or obtain information about open files
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`fwrite`	Write binary data to file

C.13 Graphics

C.13.1 2-D

`bar`	Bar graph
`grid`	Grid lines
`hist`	Histogram plot
`loglog`	Log-log scale plot
`plot`	Linear plot
`polar`	Polar co-ordinate plot
`semilogx`	Semi-log scale plot
`semilogy`	Semi-log scale plot
`text`	Text annotation
`title`	Graph title
`xlabel`	X-axis label
`ylabel`	Y-axis label
`zoom`	Zoom in and out on a 2-D plot

C.13.2 3-D

`clabel`	Contour plot elevation labels
`comet3`	Animated 3-D plot
`contour`	Contour plot in 2-D
`contour3`	Contour plot in 3-D
`mesh`	3-D mesh surface
`meshc`	3-D mesh surface with contour plot
`meshgrid`	X and Y arrays for 3-D plots
`plot3`	Line plot in 3-D
`quiver`	Quiver plot
`surf`	Shaded surface
`surfl`	Shaded surface with lighting
`view`	Rotation of 3-D figure
`zlabel`	Z-axis label

C.13.3 General

`axes`	Create axes object
`axis`	Control axis scaling and appearance
`cla`	Clear axes
`clf`	Clear current figure
`colorbar`	Display color bar (color scale)
`colormap`	Set the color look-up table (list of colormaps)
`drawnow`	Complete any pending drawing
`figure`	Create figure (graph) windows
`fplot`	Plot a function
`gca`	Get current axes handle
`gcf`	Get current figure handle
`gco`	Return handle of current graphics object
`get`	Get graphics object properties
`ginput`	Graphical input from a mouse or cursor
`gtext`	Mouse placement of text
`set`	Set graphics object properties
`subplot`	Create axes in tiled positions

Appendix D
ASCII character codes

Code	*Char*	*Code*	*Char*	*Code*	*Char*	*Code*	*Char*	*Code*	*Char*
0	(null)	26	→	52	4	78	N	104	h
1	☺	27	←	53	5	79	O	105	i
2	☻	28	∟	54	6	80	P	106	j
3	♡	29	↔	55	7	81	Q	107	k
4	◇	30	▲	56	8	82	R	108	l
5	♣	31	▼	57	9	83	S	109	m
6	♠	32	(blank)	58	:	84	T	110	n
7	•	33	!	59	;	85	U	111	o
8	◘	34	"	60	<	86	V	112	p
9	○	35	#	61	=	87	W	113	q
10	◙	36	$	62	>	88	X	114	r
11	♂	37	%	63	?	89	Y	115	s
12	♀	38	&	64	@	90	Z	116	t
13	♪	39	’	65	A	91	[	117	u
14	♫	40	(	66	B	92	\	118	v
15	☼	41	)	67	C	93	]	119	w
16	►	42	*	68	D	94	^	120	x
17	◄	43	+	69	E	95	-	121	y
18	↕	44	,	70	F	96	‘	122	z
19	!!	45	–	71	G	97	a	123	{
20	¶	46	·	72	H	98	b	124	|
21	§	47	/	73	I	99	c	125	}
22	▬	48	0	74	J	100	d	126	~
23	↨	49	1	75	K	101	e	127	∆
24	↑	50	2	76	L	102	f		
25	↓	51	3	77	M	103	g		

Appendix E
Solutions to selected exercises

Chapter 1

1.1

```
a = 3;
b = 5;
sum        =  a + b;
difference =  a - b;
product    =  a * b;
quotient   =  a / b;
```

Chapter 2

2.1 (a) comma should be replaced by decimal point
(e) asterisk should be omitted
(f) exponent must be integer
(h) comma should be replaced by decimal point

2.2 (b) decimal point not allowed
(c) first character must be a letter
(d) quotes not allowed
(h) blanks not allowed
(i) allowed but not recommended!
(k) asterisk not allowed
(l) allowed but not recommended!

2.3 (a) `p + w/u`
(b) `p + w/(u + v)`
(c) `(p + w/(u + v)) / (p + w/(u - v))`

(d) `sqrt(x)`
(e) `y^(y+z)`
(f) `x^(y^z)`
(g) `(x^y)^z`
(h) `x - x^3/(3*2)+x^5/(5*4*3*2)`

2.4
(a)
```
i = i + 1
```
(b)
```
i = i^3 + j
```
(c)
```
if e > f
   g = e
   else
   g = f
   end
```
(d)
```
if d > 0
   x = -b
   end
```
(e)
```
x = (a + b)/(c * d)
```

2.5 (a) expression not allowed on left-hand side
(b) left-hand side must be valid variable name
(c) ditto

2.6
```
a = 2;
b = -10;
c = 12;
x = (-b + sqrt(b ^ 2 - 4 * a * c)) / (2 * a)
```

2.7
```
gallons = input('Enter gallons: ');
pints = input('Enter pints: ');
pints = pints + 8 * gallons;
litres = pints / 1.76
```

2.8
```
distance = 528;
litres = 46.23;
kml = distance / litres;
l100km = 100 / kml;
disp( 'Distance   Litres used   km/L   L/100km' );
disp( [distance litres kml l100km] );
```

2.9
```
t = a;
a = b;
b = t;
```

2.10
```
a = [a b];      % make 'a' into a vector
b = a(1);
a(1) = [];
```

2.11 (a)
```
c = input('Enter Celsius temperature: ');
f = 9 * c / 5 + 32;
disp( ['The Fahrenheit temperature is: ' num2str(f)] );
```

(b)
```
c = 20 : 30;
f = 9 * c / 5 + 32;
format bank;
disp('        Celsius     Fahrenheit');
disp([c'     f']);
```

2.12
```
degrees = 0 : 10 : 360;
radians = degrees / 180 * pi;
format bank;
disp('        Degrees     Radians');
disp([degrees'     radians']);
```

2.13
```
degrees = 0 : 30 : 360;
radians = degrees / 180 * pi;
sines = sin(radians);
cosines = cos(radians);
tans = tan(radians);
table = [degrees' sines' cosines' tans']
```

2.14
```
for int = 10 : 20
  disp( [int sqrt(int)] );
end
```

2.15
```
sum(2 : 2 : 200)
```

2.16
```
m = [5  8  0  10  3  8  5  7  9  4];
disp( mean(m) )
```

2.17 x = 2.0833, a = 4.

2.18
```
% With for loop
i = 1;
x = 0;
```

```
for a = i : i : 4
  x = x + i / a;
end

% With vectors
i = 1;
a = i : i : 4;
x = i ./ a;
sum(x)
```

2.19

```
(b) n = input('Number of terms? ');
    k = 1 : n;
    s = 1 ./ (k .^ 2);
    disp(sqrt(6 * sum(s)))
```

2.21

```
r = 5;
c = 10;
l = 4;
e = 2;
w = 2;
i = e / sqrt(r ^ 2 + (2 * pi * w * l - 1 / (2 * pi * w * c)) ^ 2)
```

2.22

```
con = [200 500 700 1000 1500];
for units = con
  if units <= 500
    cost = 0.02 * units;
  elseif units <= 1000
    cost = 10 + 0.05 * (units - 500);
  else
    cost = 35 + 0.1 * (units - 1000);
  end
  charge = 5 + cost;
  disp( charge )
end
```

2.24

```
money = 1000;
for month = 1 : 12
  money = money * 1.01;
end
```

2.26

```
t = 1790 : 10 : 2000;
p = 197273000 ./ (1 + exp(-0.03134 * (t - 1913.25)));
```

```
disp([t'   p']);
pause;
plot(t,p);
```

2.27 (a)
```
r = 0.15;
l = 50000;
n = 20;

p = r * l * (1 + r / 12) ^ (12 * n) / ...
    (12 * ((1 + r / 12) ^ (12 * n) - 1))
```

2.28 (a)
```
r = 0.15;
l = 50000;
p = 800;
n = log(p / (p - r * l / 12)) / (12 * log(1 + r / 12))
```

Chapter 3

3.1 You should get a picture of tangents to a curve.

3.2 (a) 4 (b) 2

(c) The algorithm (attributed to Euclid) finds the HCF (Highest Common Factor) of two numbers by using the fact that the HCF divides exactly into the difference between the two numbers, and that if the numbers are equal, they are equal to their HCF.

3.3
```
f = input('Enter Fahrenheit temperature: ');
c = 5 / 9 * (f - 32);
disp( ['The Celsius temperature is: ' num2str(c)] );
```

3.4
```
a = input('Enter first number: ');
b = input('Enter second number: ');
if a < b
  disp( [ num2str(b) ' is larger.'] );
elseif a > b
  disp( [ num2str(a) ' is larger.'] );
else
  disp( 'Numbers are equal.' );
end
```

3.6
1. Input a, b, c, d, e, f
2. $u = ae - db$, $v = ec - bf$

3. If $u = 0$ and $v = 0$ then
   Lines coincide
   Otherwise if $u = 0$ and $v \neq 0$ then
   Lines are parallel
   Otherwise
   $x = v/u$, $y = (af - dc)/u$
   Print x, y
4. Stop.

```
a = input('Enter a: ');
b = input('Enter b: ');
c = input('Enter c: ');
d = input('Enter d: ');
e = input('Enter e: ');
f = input('Enter f: ');
u = a * e - b * d;
v = c * e - b * f;
if u == 0
  if v == 0
    disp('Lines coincide.');
  else
    disp('Lines are parallel.');
  end
else
  x = v / u;
  y = (a * f - d * c) / u;
  disp( [x y] );
end
```

Chapter 4

4.2

```
(a) log(x + x ^ 2 + a ^ 2)
(b) (exp(3 * t) + t ^ 2 * sin(4 * t)) * (cos(3 * t)) ^ 2
(c) 4 * atan(1)
(d) sec(x)^2 + cot(x)
(e) atan(a / x)
```

4.3

```
m = input('Enter length in metres: ');
inches = m * 39.37;
feet = fix(inches / 12);
inches = rem(inches, 12);
yards = fix(feet / 3);
```

```
feet = rem(feet, 3);
disp( [yards feet inches] );
```

4.5

```
a = 10;
x = 1;
k = input('How many terms do you want? ');
for n = 1 : k
  x = a * x / n;
  if rem(n, 10) == 0
    disp( [n x] );
  end
end
```

4.6

```
secs = input('Enter seconds: ');
mins = fix(secs / 60);
secs = rem(secs, 60);
hours = fix(mins / 60);
mins = rem(mins, 60);
disp( [hours mins secs] );
```

Chapter 5

5.2

```
(a) 1 1 0
(b) 0 1 0
(c) 1 0 1
(d) 0 1 1
(e) 1 1 1
(f) 0 0 0
(g) 0 2
(h) 0 0 1
```

5.3

```
neg = sum(x < 0);
pos = sum(x > 0);
zero = sum(x == 0);
```

5.7

```
units = [200 500 700 1000 1500];
cost = 10 * (units > 500) + 25 * (units > 1000) + 5;
cost = cost + 0.02 * (units <= 500) .* units;
cost = cost + 0.05 * (units > 500 & units <= 1000) .* (units - 500);
cost = cost + 0.1 * (units > 1000) .* (units - 1000);
```

Chapter 7

7.1
```
t = 1790:2000;
P = 197273000 ./ (1+exp(-0.03134*(t-1913.25)));
plot(t, P), hold, xlabel('Year'), ylabel ('Population size')
census = [3929 5308 7240 9638 12866 17069 23192 31443 38558 ...
          50156 62948 75995 91972 105711 122775 131669 150697];
census = 1000 * census;
plot(1790:10:1950, census, 'o'), hold off
```

7.2
```
a = 2;
q = 1.25;
th = 0:pi/40:5*pi;
subplot(2,2,1)
plot(a*th.*cos(th), a*th.*sin(th)), ...
     title('(a) Archimedes')    % or use polar
subplot(2,2,2)
plot(a/2*q.^th.*cos(th), a/2*q.^th.*sin(th)), ...
     title('(b) Logarithmic')   % or use polar
```

7.4
```
n = 1:1000;
d = 137.51;
th = pi*d*n/180;
r = sqrt(n);
plot(r.*cos(th), r.*sin(th), 'o')
```

7.6
```
y(1) = 0.2;
r = 3.738;
for k = 1:600
  y(k+1) = r*y(k)*(1 - y(k));
end
plot(y, '.w')
```

Chapter 8

8.1
```
balance = 1000;
for years = 1 : 10
  for months = 1 : 12
     balance = balance * 1.01;
   end
   disp( [years balance] );
end
```

8.2 (a)
```
terms = 100;
pi = 0;
sign = 1;
for n = 1 : terms
  pi = pi + sign * 4 / (2 * n - 1);
  sign = sign * (-1);
end
```

(b)
```
terms = 100;
pi = 0;
for n = 1 : terms
  pi = pi + 8 / ((4 * n - 3) * (4 * n - 1));
end
```

8.3
```
a = 1;
n = 6;
for i = 1 : 10
  n = 2 * n;
  a = sqrt(2 - sqrt(4 - a * a));
  l = n * a / 2;
  u = l / sqrt(1 - a * a / 2);
  p = (u + l) / 2;
  e = (u - l) / 2;
  disp( [n, p, e] );
end
```

8.5
```
x = 0.1;
for i = 1 : 7
  e = (1 + x) ^ (1 / x);
  disp( [x, e] );
  x = x / 10;
end
```

8.6
```
n = 6;
T = 1;
i = 0;
for t = 0:0.1:1
  i = i + 1;
  F(i) = 0;
  for k = 0 : n
    F(i) = F(i) + 1 / (2 * k + 1) * sin((2 * k + 1) * pi * t / T);
  end
  F(i) = F(i) * 4 / pi;
```

```
end
t = 0:0.1:1;
disp( [t' F'] )
plot(t, F)
```

8.8

```
sum = 0;
terms = 0;
while (sum + terms) <= 100
  terms = terms + 1;
  sum = sum + terms;
end
disp( [terms, sum] );
```

8.10

```
m = 44;
n = 28;
while m ~= n
  while m > n
    m = m - n;
  end
  while n > m
    n = n - m;
  end
end
disp(m);
```

Chapter 9

9.1

```
x = 2;
h = 10;
for i = 1 : 20
  h = h / 10;
  dx = ((x + h) ^ 2 - x * x) / h;
  disp( [h, dx] );
end
```

Chapter 10

10.1

```
function pretty(n, ch)
line = char(double(ch)*ones(1,n));
disp(line)
```

10.2
```
function newquot(fn)
x = 1;
h = 1;
for i = 1 : 10
  df = (feval(fn, x + h) - feval(fn, x)) / h;
  disp( [h, df] );
  h = h / 10;
end
```

10.3
```
function y = double(x)
y = x * 2;
```

10.4
```
function [xout, yout] = swop(x, y)
xout = y;
yout = x;
```

10.6
```
% Script file
for i = 0 : 0.1 : 4
  disp( [i, phi(i)] );
end

% Function file phi.m
function y = phi(x)
a = 0.4361836;
b = -0.1201676;
c = 0.937298;
r = exp(-0.5 * x * x) / sqrt(2 * pi);
t = 1 / (1 + 0.3326 * x);
y = 0.5 - r * (a * t + b * t * t + c * t ^ 3);
```

10.8
```
function y = f(n)
if n > 1
  y = f(n - 1) + f(n - 2);
else
  y = 1;
end
```

Chapter 15

15.1
```
heads = rand(1, 50) < 0.5;
tails = ~heads;
heads = heads * double('H');
tails = tails * double('T');
coins = char(heads + tails)
```

15.2

```
bingo = 1 : 99;
for i = 1 : 99
  temp = bingo(i);
  swop = floor(rand * 99 + 1);
  bingo(i) = bingo(swop);
  bingo(swop) = temp;
end
for i = 1 : 10 : 81
  disp(bingo(i : i + 9))
end
disp(bingo(91 : 99))
```

15.4

```
circle = 0;
square = 1000;
for i = 1 : square
  x = 2 * rand - 1;
  y = 2 * rand - 1;
  if (x * x + y * y) < 1
    circle = circle + 1;
  end
end
disp( circle / square * 4 );
```

Chapter 16

16.5

```
function x = mygauss(a, b)
n = length(a);

a(:,n+1) = b;

for k = 1:n
  a(k,:) = a(k,:)/a(k,k);  % pivot element must be 1

  for i = 1:n
    if i ~= k
      a(i,:) = a(i,:) - a(i,k) * a(k,:);
    end
  end

end

% solution is in column n+1 of a:
x = a(:,n+1);
```

Chapter 17

17.1 (a) Real roots at 1.856 and −1.697, complex roots at $-0.0791 \pm 1.780i$.
(b) 0.589, 3.096, 6.285, ... (roots get closer to multiples of π).
(c) 1, 2, 5.
(d) 1.303
(e) −3.997, 4.988, 2.241, 1.768.

17.2 Successive bisections are: 1.5, 1.25, 1.375, 1.4375 and 1.40625. The exact answer is 1.414214..., so the last bisection is within the required error.

19.3 22 (exact answer is 21.3333).

19.4 After 30 years the exact answer is 2 117 ($1000\,e^{rt}$).

19.6 The differential equations to be solved are

$$\begin{aligned} dS/dt &= -r_1 S, \\ dY/dt &= r_1 S - r_2 Y. \end{aligned}$$

The exact solution after 8 hours is $S = 6.450 \times 10^{25}$ and $Y = 2.312 \times 10^{26}$.

17.8
```
function s = simp(f, a, b, h)
x1 = a + 2 * h : 2 * h : b - 2 * h;
sum1 = sum(feval(f, x1));
x2 = a + h : 2 * h : b - h;
sum2 = sum(feval(f, x2));
s = h / 3 * (feval(f, a) + feval(f, b) + 2 * sum1 + 4 * sum2);
```

With 10 intervals ($n = 5$), the luminous efficiency is 14.512725%. With 20 intervals it is 14.512667%. These results justify the use of 10 intervals in any further computations involving this problem. This is a standard way of testing the accuracy of a numerical method: halve the step-length and see how much the solution changes.

17.9
```
% Command Window
beta = 1;
```

```
ep = 0.5;
[t, x] = ode45(@vdpol, [0 20], [0; 1], [], beta, ep);
plot(x(:,1), x(:,2))

% Function file vdpol.m
function f = vdpol(t, x, b, ep)
f = zeros(2,1);
f(1) = x(2);
f(2) = ep * (1 - x(1)^2) * x(2) - b^2 * x(1);
```

This page intentionally left blank

Index

%, 34
%e format specifier, 72
%f format specifier, 72
%g format specifier, 72
&, 128
', 130, 144
.', 130
.*, 40
./, 40
.^, 40
:, 9, 144
;, 7, 42
~, 128
π, 8
  estimation of, 337
0-1 vector, 132
3-D plot, 181

abs, 70, 105
acos, 105
acosh, 105
algorithm, 86–90, 92–93, 95, 07, 99, 101–102
all, 133, 151
  example, 134
AND, 128
angle, 70
animation, 281
  comet3, 183
ans, 26, 41
any, 133
  example, 134
apostrophe, 144
  included in string, 160
area, 192
argument
  input, 231
  input, pass by reference, 238
  input, pass by value, 238
arithmetic operator, 37
array, 9, 25, 153
  2-D, 141
  cell, 267
  multi-dimensional, 156
  string, 161
Array Editor, 27
array operation, 40, 153
ASCII code, 162
asin, 105
asinh, 106
aspect ratio, 70
assignment statement, 7, 42
atan, 106
atan2, 70, 106
atanh, 106
axis, 174
  equal, 70
  normal, 70
axis limits, 175
azimuth, 191

bacteria division, 331
bacteria growth, 338, 370
balance delimiters, 46
bar, 192
bar graph, 260
base workspace, 236
best fit, 351
binary to decimal conversion, 170
binomial coefficient, 205
Bisection method, 362
bit, 26
bitwise logical operations, 128
blanks, 164
break, 208, 215
breakpoint, 244
Bubble Sort, 261
bug, 222
byte, 26

calendar, 9
camel caps, 25
Cantilever beam, 307, 308, 309, 310, 311, 326
cartesian co-ordinates, 179
case, 69
case sensitivity, 25
catch, 228
ceil, 106
cell, 267, 268
cell array, 267
celldisp, 270
cellplot, 270

Cervantes, 170
chaos, 204, 283, 376
  butterfly, 379
  trajectory, 379
char, 162
cla, 178
clc, 9
clear, 26
clear M-file, 236
clf, 178
clipboard, 17
clock, 57, 106, 296
colon operator, 9, 29, 144
  initialization, 29
  precedence, 38
colorbar, 286
colormap, 285
colour, 175
comet3, 183
command, 43
command line, 7
command line editing, 7
Command Window, 5
command/function duality, 242
comment, 34
compass, 193
compiler, 21
complex conjugate transpose, 70
complex numbers, 69
concatenation of strings, 161
condition estimator, 352
conj, 70
Contents.m, 235
context menu, 20
continuation of statement, 35, 42
continue, 215
contour, 186
contour plot, 13
contour3, 187
conversion, binary to decimal, 170
cos, 106
cosh, 106
cot, 106
Crank-Nicolson method, 382
csc, 106
cumsum, 106
current directory, 15
Current Directory browser, 20
curve fitting, 386
cut and paste, 17

data types, 36
date, 9, 106
deal, 266
deblank, 164
Debugger, 243
debugging, 222
degrees to radians, 8
del2, 288
delimiter highlighting, 46
demo, 12
desktop, 15
  default, 15
det, 160
determinate repetition, 205
diag, 152
diary, 32, 118
dice, 127
diff, 368
differential equation, 369
disp, 47
  scale factor, 50
division, matrix, 352
docking, 15
dot product, 40
dot-transpose operator, 130
double, 37
double-precision, 36
doubling time of an investment, 210
drawnow, 283

editing a plot, 279
eig, 160
eigenvalue, 344
eigenvector, 345
Electric circuit, 307, 308, 311, 314
element-by-element operation, 40
ellipse, equation of, 203
ellipsis, 35, 42
elseif, 64
empty, 29
  array, 133
end, 59, 146
end of file, 116
eps, 124
error
  fatal, 222
  logical, 226
  rounding, 226
  run-time, 243
  statement, 228
  syntax, 222
  trapping, 166
errorbar, 194
escape code, 72
etime, 57
Euler's method, 369
eval, error trapping, 166
exclusive OR, 128
exist, 133
exp, 106
explicit list, initializing vectors, 29
expm, 160, 374

exponent, 36
exponential fit, 351
exponential growth, 370
exponential matrix, 374
exporting ASCII data, save, 110
exporting binary data, save, 111
expression, 35–37, 41
  ans, 41
  display suppressed, 41
  evaluation, 165
  logical, 60, 127
  response to input, 75
`eye`, 150
`ezcontourf`, 194

factorial, 53
`false`, 121
`feather`, 195
`feof`, 116
`feval`, 240
Fibonacci numbers, 250
field, 264
fieldnames, 265
`figure`, 10, 178
figure window, 10, 176
file I/O functions, 73
file identifier, 114
`fill`, 195
`find`, 133
`findobj`, 276
finite difference scheme, 371, 381
`fix`, 106
fixed point, 35, 49
`fliplr`, 152
`flipud`, 152
floating point, 35, 49
`floor`, 55, 107
`fopen`, 114
`for`, 54
  command line version, 56
  common error, 59
  index, 54
  iteration count, 55
  more general, 56
  most general form, 153
  nested, 154, 208
  replaced by vectorization, 56
  used for determinate loop, 216
`format`, 49
format specifiers, 72
formula, vectorization, 44
Fourier series, 219
`fplot`, 181
`fprintf`, 71
  escape code, 72
  string, 163
fractal, 204
`fread`, 116
free fall with and without friction, 307, 308, 314, 315, 318, 319, 320, 321, 322, 323
frequency distribution, 260
`fseek`, 117
`ftell`, 117
`full`, 356
full matrix, 354
function, 43, 234
  handle for passing as argument, 240
  help, 235
  MATLAB Function Reference, 105
  name resolution, 243
  P-code file, 239
  private, 239
  subfunction, 239
function functions, 242
function M-file, 230
  Newton's method, 232
function/command duality, 242
`fzero`, 364

Gauss reduction, 358
Gaussian random number, 336
`gca`, 274
`gcbo`, 296
`gcf`, 274
`gco`, 274
`get`, 274
Gibbs phenomenon, 219
`ginput`, 178
`global`, 236
`gradient`, 188
graph, 10
graphical user interface, 292
graphics
  2-D, 171
  3-D, 181
  3-D rotation, 190
  axis limits, 175
  contour plot, 186
  cropping a surface, 187
  easy-to-use commands, 172
  editing a plot, 279
  line type, 174
  logarithmic plot, 178
  mesh surface, 183
  multiple plots, 173, 176
  on GUI, 301
  plotting symbol, 174
  polar co-ordinates, 174
  rapidly changing functions, 180
  saving, 289
  vector field, 188
`grid`, on graph 173
`gtext`, 173, 258

guessing game, 208
GUI, 292
  handles structure, 300
  plotting graph on, 301
guide, 292

H1 line, 235
Halley's comet, 203
Handle Graphics objects, 272
handle
  function, 240
  graphics object, 272
handles structure in a GUI, 300
Help, 15
help, 13
Help browser, 15
Help Navigator, 105
help for a function, 235
helpwin, 105
hexadecimal, 50
hidden name, 73, 225, 243
hist, 196
histogram, 260
hold, 174

i, 69
I/O functions, 73
identity matrix, 150
if, 60
  command line version, 60
  elseif, 64
  nested, 67
  vectorized with logical vector, 135
if-else, 63
  command line, 63
ill conditioning, 352, 357
imag, 70
Import Wizard, 112
importing ASCII data, load, 111
importing binary data, load, 112
importing data, Import Wizard, 112
inclusive OR, 128
indeterminate repetition, 205, 208
index of for, 54
Inf, 7
infinity, 7
initialization, colon operator, 29
  explicit list, 29
inline object, 231
input, 75
  argument, 231
  string, 160
  vector, 75
input argument 231
int2str, 164
interpolation, 366
interpreter, 21

inv, 160
ischar, 164
isempty, 133
isglobal, 236
isinf, 134
isnan, 134
isothermal, 186
iteration count, 55

j, 69

Knuth, 212

lasterr, 225
  error trapping, 166
Launch Pad, 12
least squares, 351
left division, 37, 349, 352
Legendre polynomial, 250
length, 107
Leslie matrix, 343
lighting, 288
limit of sequence, 53
line numbers, 244
line style, 174
linear equations
  over-determined, 350
  residual, 350
  solution, 348
  under-determined, 351
linspace, 30
list, sorting, 261
load, 111, 112
LODE, 374
log, 107
log10, 107
logarithmic plot, 178
logarithmic spiral, 201
logical error, 226
logical expression, 60, 121, 127
  vector, 122
logical function, 133
logical operator, 65, 127
  precedence, 128
logical vector, 122
  subscripting with, 131
loglog, 179
lookfor, 13, 235
loop
  determinate, 205
  indeterminate, 208
Lorenz, 376
Lotka-Volterra model, 380
lower, 164
lu, 160

M-file, 19
  function, 230
  function passed as argument, 240
macro, 165
magic square, 13, 151
mantissa, 36, 49
marker, 174
Markov chain, 345
matrix, 27, 31, 141
  array operation, 153
  creating, 143
  deleting rows and columns, 148
  division, 352
  duplicating rows and columns, 148
  enlarging, 144
  exponential, 374
  exponentiation, 159
  full, 354
  Leslie, 343
  multiplication, 157
  operation, 40, 157
  plot, 171
  sparse, 354
  subscript, 144
  transpose, 32, 144
  visualization, 189
`max`, 107, 264
`mean`, 107
`menu`, 215
`mesh`, 184
mesh surface, 183
`meshc`, 187
`meshgrid`, 183
`meshz`, 197
Mexican hat, 13
`min`, 107, 264
`mlock`, 237
Monte Carlo estimation of π, 337
`more`, 48, 213
multi-dimensional array, 156
multiple lines in Command Window, 18
multiple plots, 176
  on same axes, 173
`munlock`, 237

name hiding, 225, 243
`NaN`, 7
`nargin`, 238
`nargout`, 239
nested `for`s, 154, 208
nested `if`s, 67
Newton quotient, 228, 249, 367
Newton's method, 51
  in general, 359
normally distributed random number, 336
NOT, 128
not-a-number, 7
`num2str`, 48, 164
number, 35
numerical differentiation, 367
numerical integration, 364
numerical method, 359
  Bisection method, 362
  differential equation, 369
  differentiation, 367
  Euler's method, 369
  integration, 364
  Newton quotient, 367
  Newton's method, 359
  partial differential equation, 381
  predictor-corrector, 373
  Runge-Kutta method, 375
  Simpson's rule, 366
  solution of equations, 359
  Trapezoidal rule, 365
numerical solution
  differential equations, 369
  equations, 359

object, 270
  Handle Graphics, 272
object-oriented programming, 270
`ones`, 149, 162
operand, 37
operating system, shelling out to, 75
operation
  array, 40
  element-by element, 40
  matrix, 157
operator precedence, complete list, 129, 395
operator
  arithmetic, 37
  colon, 144
  dot-transpose, 130
  left division, 349
  logical, 65, 127
  precedence, 37, 129
  relational, 60
  separate with brackets, 131
  transpose, 130
OR, 128
  exclusive, 128
output, 47
  paging, 48
  suppressed with semi-colon, 42
  to a file with fprintf, 73
output argument, 233
over-determined system, 350

P-code file, 239
paging output, 48, 213
palindrome, 169

partial differential equation, 381
pascal, 151
pause, 191
pcode, 239
peaks, 192
persistent variable, 236
pi, 8
pie, 197
pitfall, 225
plot, 171
  3-D version, 182
  logarithmic scale, 178
  polar co-ordinates, 179
plot edit mode, 279
plot3, 182
plotedit, 279
plotyy, 174
Poisson process, 333
polar, 180
polar co-ordinates, 179
polynomial fitting, 386
pow2, 107
pre-allocation, 253
precedence, 129
  arithmetic operators, 37
  colon operator, 38
  logical operator, 128
  of operators, complete list, 130, 395
  transpose, 39
predator-prey model, 380
predictor-corrector method, 373
Preferences, 50
prime number
  largest known, 212
  test, 211
private function, 239
prod, 108
Profiler, 240
program, 16
programming style, 76
projectile trajectory, 212
projectile (without friction), 89, 90, 92–94
projectile with friction, 307, 308, 323, 324, 327
Property Editor, 181
pseudo-code, 98
pseudo-random number, 108

qr, 160
quad, 367
quadratic equation, 99
  structure plan, 99
quadrature, 367
quiver, 188
quote in string, 48

radio-active decay, 340
RAM, 21
rand, 108, 126, 149, 328
  seeding, 329
  uniform, 328
randn, 336
random access memory, 21
random number, 31
  counting, 126
  generation, 328
  normal, 336
  uniform, 328
random walk, 260, 331, 345
rat, 50
real, 70
realmax, 36, 108
realmin, 36, 108
recall, smart, 7
recursion, 247
redimensioning, 254
reference, pass by, 238
relational operator, 60
relative accuracy, 36
rem, 108
removing elements of an array, 31, 132
repmat, 148
reshape, 147
reshaping a matrix, 147
residual, 350
ribbon, 198
right division, 353
rmfield, 266
rolling dice, 127, 330
root of an equation, 359
roots, 364
rot90, 152
rounding error, 226, 367
run-time error, 243
Runge-Kutta method, 375

save, 73, 110, 111
saving a file, 19
saving a graph, 289
scalar, 9
  expansion, 40
  product, 40
scientific notation, 36
  display, 49
scope, 236
script file, 19
  hidden by variable of same name, 73
  recommended way to run, 20
search path, 74
seeding rand, 329
semi-colon, 7
semilogx, 179
semilogy, 178

sensitivity analysis, 357
separator, 42
series for π, 217
set, 275
Shakespeare, 170
shelling out to the operating system, 75
Simpson's rule, 366
simulation, 127, 260
  bacteria division, 331
  bacteria growth, 338
  game of eights, 339
  π, estimation of, 337
  radio-active decay, 340
  random walk, 331, 348
  rolling a die, 330
  spinning coins, 329
  traffic flow, 333
single-precision, 36
size, 26
smart indenting, 55
smart recall, 7
sort, 261, 263
sort, bubble, 261
sound, 208
sparse, 354, 356, 383
sparse matrix, 354, 383
spinning coins, 329
spiral
  Archimedes, 201
  logarithmic, 201
sprintf, 164, 258
spy, 190, 356
sqrt, 8
stable age distribution, 345
stairs, 199
statement, 7, 42
  assignment, 42
  continuation, 42
stem3, 199
strcmp, 164
string, 48, 141
  array of characters, 161
  ASCII code, 162
  comparison, 163
  concatenation, 161
  containing quote, 48
  fprintf, 163
  including apostrophe, 160
  input, 160
  two-dimensional, 164
struct, 265
structure, 264
structure plan, 18, 86–89, 91, 98–102
  quadratic equation, 99
structured programming, 100
style, 76
subfunction, 239
subplot, 176
subscript, 31, 144
  logical vectors, 131
  out of range, 144
  vector, 131
subscripting with logical vectors, 131
sunflower head, 202
surf, 184
surfc, 187
svd, 160
switch, 68
syntax error, 222
syntax highlighting, 55

tab completion, 11
Taylor series, exponential, 249
TEX characters, 280
text label on graph, 173
text macro, 165
tic, 57
tiling, 148
title, 173
toc, 57
top down programming, 98
traffic flow, 333
transpose, 30, 32, 130, 144
  complex conjugate, 70
  precedence, 39
Trapezoidal rule, 365
tridiagonal matrix, 383
tril, 152
triu, 152
true, 121
truecolor, 288
truncation error, 386
try/catch, 167, 228
type, 19

uicontrols, 293
under-determined system, 351
undocking, 15
unhiding a name, 74
uniformly distributed random number, 328
update process, 251
upper, 164

value, pass by, 238
Van der Pol's equation, 389
varargin, 239, 269
varargout, 269
variable, 7, 21, 24
  global, 236
  hides script of same name, 73
  persistent, 236

variable (*continued*)
  scope, 236
  string, 160
  termed arrays by MATLAB, 25
vector, 9, 27
  0-1, 132
  colon operator, 29
  explicit list, 28
  incompatible sizes, 225
  input, 75
  pre-allocation, 253
  redimensioning, 254
  removing element, 132
  scalar product, 40
  subscript, 31, 131
  transpose, 30
vectorization, 44
  instead of `for`, 56
`view`, 190
visualization of matrix, 189

`waterfall` 200
`while`, 209
  command line form, 210
  used for indeterminate loop, 216
  vector condition, 210
`who`, 26
`whos`, 26
`why`, 14
workspace, 25
  base, 236
  function, 236
Workspace browser, 15, 27

`xlabel`, 173
`xor`, 128

`ylabel`, 173

Zeller's congruence, 169
zero of a function, 359
`zeros`, 149
`zlabel`, 182
`zoom`, 387